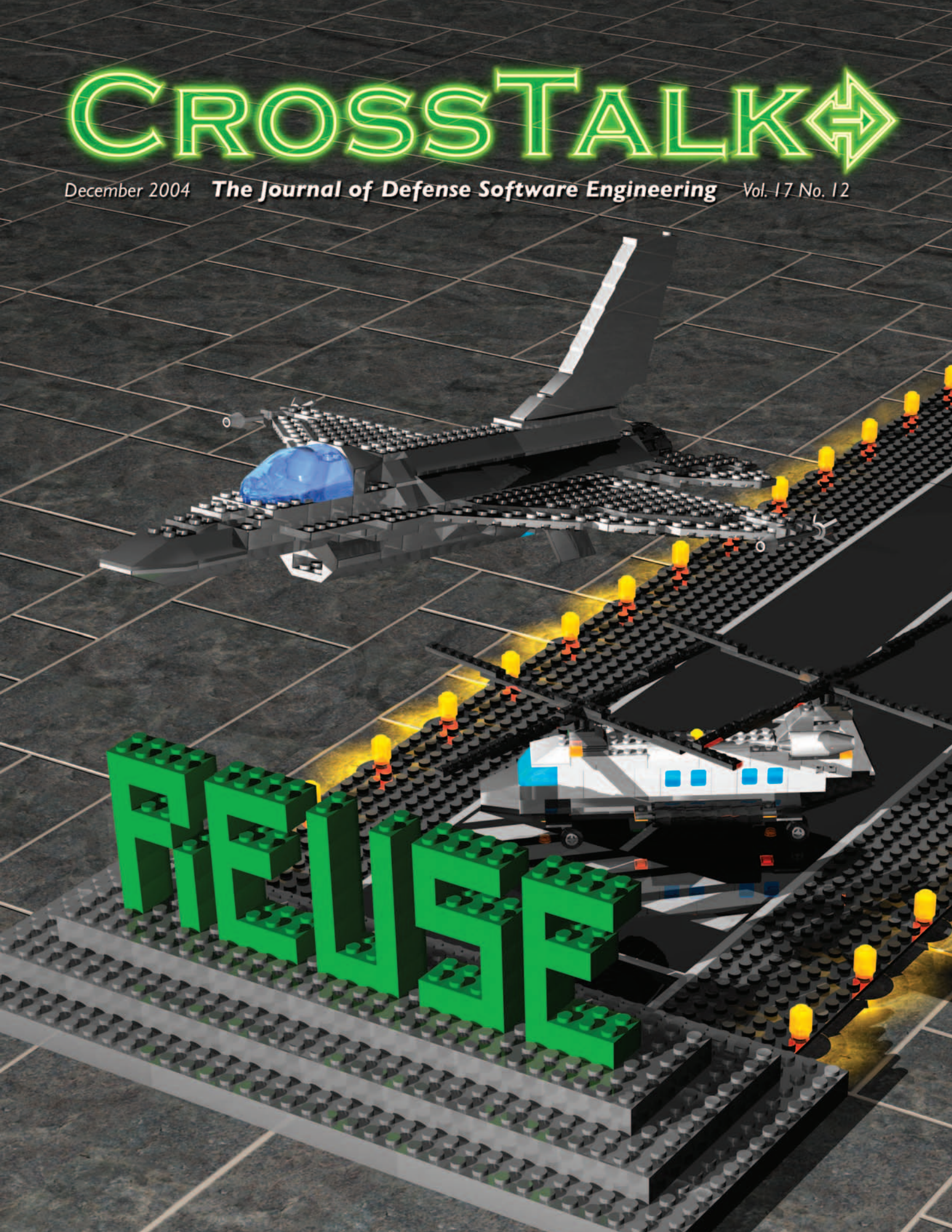
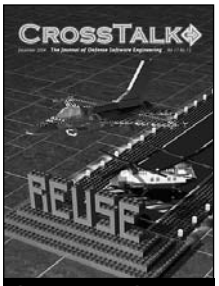

CROSSTALK
December 2004 The Journal of Defense Software Engineering Vol. 17 No. 12
REUSE

Reuse



ON THE COVER
Cover Design by Kent Bingham.

Departments

The CrossTalk staff would like to wish you and yours the very best this holiday season and the happiest of New Years.

CROSSTALK

OC-ALC/ MAS Co-Sponsor	*Kevin Stamey*
OO-ALC/MAS Co-Sponsor	*Randy Hill*
WR-ALC/MAS Co-Sponsor	*Tom Christian*
Publisher	*Tracy Stauder*
Associate Publisher	*Elizabeth Starrett*
Managing Editor	*Pamela Palmer*
Associate Editor	*Chelene Fortier-Lozancich*
Article Coordinator	*Nicole Kentta*
Creative Services Coordinator	*Janna Kay Jensen*
Phone	(801) 775-5555
Fax	(801) 777-8069
E-mail	crosstalk.staff@hill.af.mil
CrossTalk Online	www.stsc.hill.af.mil/crosstalk

Oklahoma City-Air Logistics Center (OC-ALC), Ogden-Air Logistics Center (OO-ALC), and Warner Robins-Air Logistics Center (WR-ALC) MAS Software Divisions are the official co-sponsors of CrossTalk, The Journal of Defense Software Engineering. The MAS Software Divisions and the Software Technology Support Center (STSC) are working jointly to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.

The STSC is the publisher of CrossTalk, providing both editorial oversight and technical review of the journal.

S T S C

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address. You may e-mail us or use the form on p. 22.

OO ALC/MASE
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CrossTalk editorial board prior to publication. Please follow the Author Guidelines, available at <www.stsc.hill.af.mil/crosstalk/xtlkguid.pdf>. CrossTalk does not pay for submissions. Articles published in CrossTalk remain the property of the authors and may be submitted to other publications.

Reprints: Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with CrossTalk.

Trademarks and Endorsements: This Department of Defense (DoD) journal is an authorized publication for members of the DoD . Contents of CrossTalk are not necessarily the official views of, or endorsed by, the U.S. government, the DoD, or the STSC. All product names referenced in this issue are trademarks of their companies.

Coming Events: Please submit conferences, seminars, symposiums, etc. that are of interest to our readers at least 90 days before registration. Mail or e-mail announcements to us.

CrossTalk Online Services: See <www.stsc.hill.af.mil/crosstalk>, call (801) 777-7026, or e-mail <stsc.webmaster@hill.af.mil>.

Back Issues Available: Please phone or e-mail us to see if back issues are available free of charge.

Reuse: A Maturing Practice

CrossTalk has been covering the many aspects of reuse since the early 1990s. Those aspects include publishing policies, initiatives, techniques, best practices, and lessons learned from those in the field who have been researching and practicing reuse. During this time, and as shown by this issue's full set of articles, reuse has become more widely accepted and continues to mature in the defense software community.

In the past, many teams often discarded reuse on their projects primarily because they had no foundation, strategy, or process to employ reuse. As a programmer, I remember many times wondering if I was caught in a *duplication-of-effort* trap. Was the code I was generating already developed and sitting in a library somewhere? Who do I ask? Where do I look? Much has changed, especially with reuse applications such as commercial off-the-shelf software, government off-the-shelf software, network-centric architectures, and open source software. With the Web and portal environments now at many programmers' fingertips, information on reusable components and artifacts has never been more available and accessible.

The management of projects employing reuse is also maturing. With managers' interests peaking in the cost of reuse, we begin this month's issue with *An Economic Analysis of Software Reuse*, by Dr. Randall W. Jensen. This article presents results from a simplified economic model that predicts software product development costs in an environment containing reused software components. Results from Jensen's analysis are independent of any software estimating tools or models. The specific reusable component types considered in the analysis include requirements, design, code, and validated code. Additional information on the cost of reuse is presented in *Estimating and Managing Project Scope for Maintenance and Reuse Projects* by William Roetzheim. This author discusses quantitative approaches to estimating scope and effort for maintenance, enhancement, and reuse projects.

Next, Dr. Kelvin Nilsen brings us *Using Java for Reusable Embedded Real-Time Component Libraries***.** Nilsen discusses how Java has progressed and now supports mission-critical, real-time systems. With its portability features and its appeal to programmers, Java is helping to lay the foundation for a reusable software component industry. Nilsen also discusses how standards are being developed to help encourage competitive pricing and innovation among Java technology vendors while assuring portability and interoperability of real-time components.

Reuse is also discussed in our next article, *Separate Money Tubs Hurt Software Productivity* by Dr. Ronald J. Leach. This author shows how cost and quality improvements are rarely simultaneously achieved due to the common management and project accounting practice of *every tub on its bottom* where a *tub* of money is utilized solely for one project. The author suggests simple changes to funding approaches to help achieve key objectives of *more* software *better*, *cheaper*, and *faster*.

Our final article this month is *Reuse and DO-178B Certified Software: Beginning With Reuse Basics* by Hoyt Lougee. This article presents a good summary of reuse basics and discusses the less publicized safety benefits of reuse. In the DO-178B guidance, objectives and activities that must be performed in developing and verifying airborne software systems are defined. Adherence to DO-178B is causing many avionics manufacturers to turn to reuse. This article is informative for all readers interested in reuse even if you are not concerned with DO-178B.

As we put the wraps on our 17th volume, I hope you find this issue a good source of continuous learning on the subject of reuse. Don't forget to check out CrossTalk's Volume 17 Article Index (see page 29) highlighting the many other subjects and articles published in 2004. And finally, I close by giving a special thanks to our readers, authors, and new co-sponsors: the United States Air Force's Air Logistics Centers and their three Software Divisions. Their sponsorship commitment is a great gift this season to all of us in the defense software community.

On behalf of the CrossTalk staff, I wish you a safe and wonderful holiday season.

Tracy L. Stauder

Tracy L. Stauder
Publisher

An Economic Analysis of Software Reuse©

Dr. Randall W. Jensen
Software Technology Support Center

This article presents a simplified economic analysis of the cost of software reuse. The reuse definition used here includes both commercial off-the-shelf (COTS) and existing software from an upgraded platform. The results are independent of software estimating tools or models. The model used in this analysis relates the cost of software development to the reused software level and the costs of developing and maintaining the software components. COTS software is a special case of reuse described in this article.

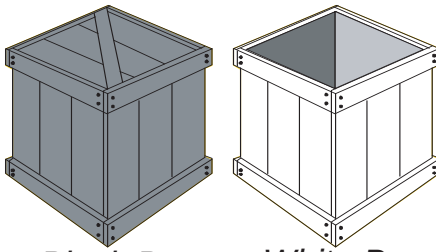
Current software projects tend to maximize reusable component use and minimize development product size. There are significant advantages to using reusable components:

- Lower development time and effort through using existing, supported components.
- Reduced risk through using proven field-tested components.

High customer demand, reduced software development budgets, and a competitive software market drive the need for reusable software. The downside to reusable software is a high development cost, and the significant cost of integrating reusable components into software products. These integration costs can be devastating if components are inadequate, poorly defined and documented, or not quite compatible with the application.

Gaffney and Durek [1] published the first economic analysis report of this type in 1988. Marion Moon and I (while at Hughes Aircraft Company) in 1989 initiated an economic analysis in response to [1]. Unfortunately, the project was shelved before completion due to higher priority tasks. The interest in reuse cost and the need for the economic analysis has continued to increase since that time. Meanwhile, the acronym COTS (commercial off-the-shelf) has largely replaced the term reuse, but the costs associated with reuse have remained the same.

Figure 1: *Black Box Versus White Box*



The analysis in this article focuses on the primary measurable costs associated with reuse, but does not consider several hard-to-predict costs:

- Vendor upgrade release to reusable component.
- Vendor discontinuing component support.
- Component requirement or capability changes.

> *"High customer demand, reduced software development budgets, and a competitive software market drive the need for reusable software."*

- Cost of component evaluation and selection.
- Understanding component function or external interfaces.

The objective of this analysis is to show there are significant cost impacts of software reuse without considering the costs associated with the less-defined factors listed above. The analysis results show significant and somewhat optimistic cost impacts.

Black-Box Phenomenon

The concept of a black box is widely used in system and hardware design. A black box is a system (or component, object, etc.) with known inputs, known outputs, a known input-output relationship, and unknown or irrelevant contents. The box is black; that is, the contents are not visible as shown in Figure 1. The black-box concept is particularly important where components, or objects, are used without the engineering, implementation, integration, and test costs associated with their development.

A white box is a component that requires knowledge of the box contents to be used. A software component becomes a white box when either of the following conditions exist:

- A modification is required to meet software requirements.
- Documentation that is more extensive than an interface or functional description is required before the component can be incorporated into the software system.

Black-box component behavior is characterized in terms of an input set, an output set, and a relationship (hopefully simple) between the two sets. Behavior must be uniquely determined for all input and output combinations. Behavior must be stable and reliable. Once behavior becomes unstable, unreliable, or slightly different than the project needs, the component becomes a white box. Effective size S_e is a major difference between black box and white (or gray) box components from an estimating point of view. The effective size of the software within the black box is zero. Prying the lid off the box has serious consequences in terms of effective size.

Reusable software in this analysis satisfies the black box component definition.

First-Order Reuse Cost Model

There are some conditions we need to assume in this economic analysis:

- The software must satisfy the black-box requirements at the level of abstraction being applied; that is, the reused software satisfies the required performance requirements without modification.
- User knowledge is expert within the scope of reuse.
- Documentation is adequate for the

reuse needs.
- Cost of reusable component selection, evaluation, and purchase are ignored.
- The product is *rock solid*; that is, no maintenance is required, and no vendor upgrades will be made.

A software system contains three categories of source code: new S_n; original S_o, including both modified and *lifted* (lifted is a term for unchanged original code); and reused S_r as shown in Figure 2. The effective size S_e used in most software cost and schedule estimates is an adjusted combination of the new and modified source code similar to the equation:

$$S_e = S_n + S_o (A_d \times F_d + A_i \times F_i + A_t \times F_t) \quad (1)$$

where,

A_d = Design activity, A_i = Integration activity, A_t = Test activity and $A_d + A_i + A_t = 1$. The parenthetical factor is a weighted combination of relative efforts from the design (F_d), implementation (F_i), and test (F_t) activities. More thorough discussions of effective size can be found in the references.

The remainder of the system consists of one or more reusable components. Since reusable components are black boxes that have no accessible size, we cannot directly apply an effective size equation to form an estimate.

For our purposes, we are going to assume the relative reusable component(s) size can be derived by estimating the size of the reusable component built from scratch S_r as:

$$R = \frac{S_r}{(S_e + S_r)} \quad (2)$$

where,

R is the portion (fraction) of the system to be implemented by reusable source code.

The first-level economic model of software reuse begins with the assumption that the cost of software development *C* for a product relative to the cost of all new source code can be given by the equation:

$$C = 1(1 - R) + bR$$

or

$$C = 1 + R(b - 1) \quad (3)$$

where,

Equation Legend

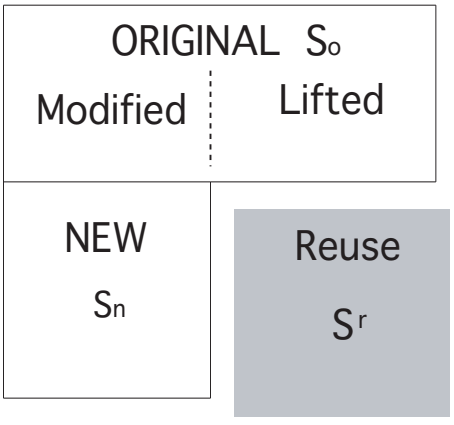
a	Reusable component development cost relative to the cost of non-reusable development from scratch.
b	Relative cost of incorporating reusable components into developed system.
C	Relative cost of software development.
F	Relative COTS acquisition cost relative to the cost of non-reusable development from scratch.
n	Number of uses over which the reusable product cost is amortized.
R	Portion (fraction) of the system to be implemented by reusable source code.
F_d	Design effort relative to design from scratch.
F_i	Implementation effort relative to implement from scratch.
F_t	Test effort relative to test from scratch.
S_{COTS}	Estimated size of an internally developed COTS replacement.
S_e	Effective source size for devlopment.
S_n	New source code to be added to system.
S_o	Original source size from pre-existing system.

C=1 is the cost of developing a system from scratch. The factor *b* represents the cost of incorporating reused components into the system relative to developing the components from scratch. The term (*1-R*) represents the fraction of new and/or modified source code. This model was first published by Gaffney and Durek [2] and is the basis of this analysis.

Reuse can occur at several levels: requirements, design, code, and validated code. Using the relative design, implementation, and integration factors from Sage [3], we find the relative cost for each development activity given by Table 1 (see page 6). The relative cost values based on Constructive Cost Model (COCOMO) [4]/Revised Intermediate COCOMO (REVIC) [5] are also included in the table for comparison.

The reusable component type (abstraction) determines the relative cost factor *b* in Equation (3). The specific reusable component types considered in this analysis are requirements, design, code, and validated code. The activities to develop these are defined as follows:
- **Requirements.** Includes the analysis and synthesis of software requirements. The product resulting from this activity is the Software Requirements Specification (SRS). The activity is often terminated with a software requirements review (SRR). The definition of system requirements, if present, is not part of the software requirements analysis activity.
- **Design.** Includes the architecture and detailed design of the software product. The resulting product of this activity is a detailed product specification containing both the architecture and component specifications. The activity is usually terminated by a

Figure 2: *Software System Architecture for Reuse Analysis*



Activity	Activity Code	Relative Cost, Sage	Relative Cost, REVIC/COCOMO
Requirements	Req	0.07	0.07
Design	Des	0.38	0.41
Implementation	Imp	0.23	0.26
Integration and Test	Test	0.32	0.26

Table 1: *Relative Costs of Development Activities*

detailed design review (often referred to as a critical design review, or CDR).

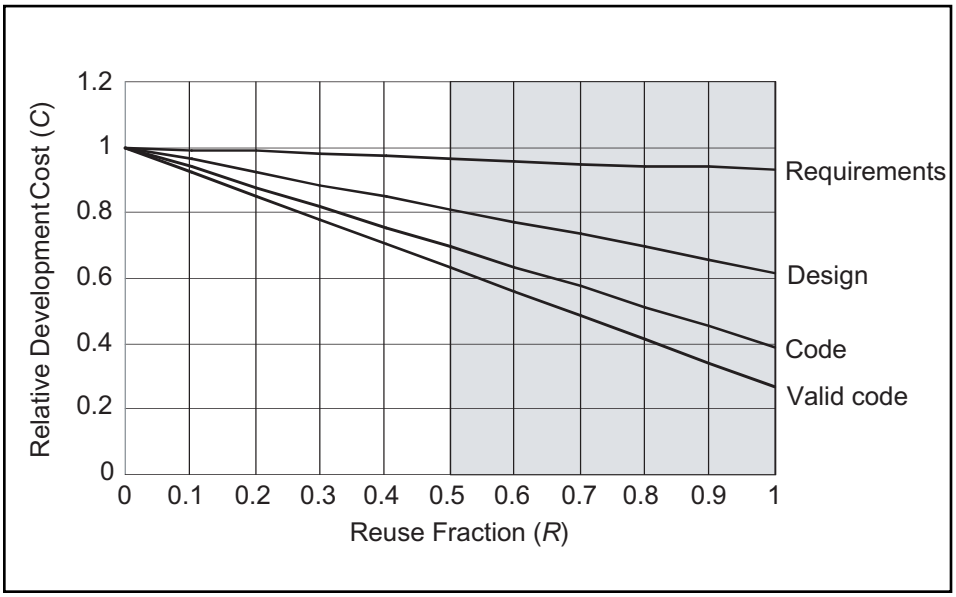
- **Implementation.** Implements the detailed software design in the specified programming language(s), and verifies the individual component (unit) performance to the requirements specified in the detailed product specification.
- **Integration and Test.** Integrates (combines) the tested software components into a larger structure that represents the software product. The activity may contain one or more computer programs. The components are individually defined by formal requirements and interface specifications. The activity usually culminates with a qualification test that evaluates performance per the software requirements specification for the product. The test is usually conducted at the development facility with controlled test data.
- **Regression Test.** Integrates previously validated components into a larger software product structure. This activity may contain one or more computer programs. The regression test activity ends with satisfactory completion of the final qualification test that evaluates product performance per the software requirements and interface specification. Regression test usually reduces the early integration tests required by the integration and test activity.

Table 2 combines the relative activity costs from Table 1 to provide the relative component reuse cost *b* values. For example, the design, implementation, and test activities must be completed to incorporate a requirements reuse component. The incorporation cost is the sum of the relative activity costs, or $b = 0.93$. If the code is reused, the requirements effort for this component of the new system must still be performed. Also note the relative integration cost *b* for validated code is assumed to be *0.27* instead of the *0.32* value assumed by Sage for normal integration and test. This decrease accounts for the reduced testing requirements of validated code.

Table 2: *Relative Reuse Cost*

Component Type	Activities to Be Completed	Relative Reuse Cost (*b*)	Relative Development Cost
Requirements	Design, Implementation, Test	0.93	0.07
Design	Requirements, Implementation Test	0.62	0.45
Code	Requirements, Test	0.39	0.68
Validated Code	Requirements, Test (Regression)	0.27	1.00

Figure 3: *Relative Development Cost vs. Reuse Fraction By Reuse Component Type*



The cost relationship between relative development cost *C* and the percent of reusable software *R* is illustrated graphically in Figure 3. The graph shows the relative costs for each reuse component type (*b* values defined in Table 2). Reuse percentages greater than 50 percent are uncommon and are highlighted with a gray background in the figure. The simple cost model shows that the maximum cost reduction for a software system containing 50 percent COTS software (validated code *b=0.27*) is only about 37 percent (relative development cost is 63 percent). If 100 percent validated code reuse were possible, the software still costs 27 percent of the cost required to build the software system from scratch due largely to regression testing.

Higher Order Cost Model

The first issue that must be considered in developing the reuse cost model is the cost of the reusable component. Incorporating the development cost into the economic model yields:

$$C = (1 - R) \times 1 + (b + \frac{a}{n})R \qquad (4)$$

where,

a is the reusable component development cost relative to the cost of non-reusable development from scratch, and *n* is the number of uses over which the reusable product cost is amortized. The model then becomes:

$$C = (b + \frac{a}{n} - 1)R + 1 \qquad (5)$$

The relative component development cost is at least equal to the non-reusable software development cost. The development cost could double when the effort required to make the component more robust is considered. For this analysis we assume the relative component development cost *a* is in the realistic range $1.0 \le a \le 2.0$. The factor a/n in the cost model accounts for the amortized cost of providing the reusable software to this project. Equation (5) shows that as long as the coefficient is $b + a/n \le 1$, reuse will provide a positive cost incentive; that is, $C \le 1$. We will look at the cost incentive

further in the next section.

The factor *a* can also be used to relate the relative cost of purchasing, or otherwise acquiring, the reusable component(s) for the project. In this case, the component acquisition cost *a* is in the range $0.0 \le a \le 2$ where the reusable component acquisition relative cost includes evaluation, selection, and procurement. As acquisition cost approaches development cost, acquisition becomes less attractive.

The reusable component acquisition cost can be treated in a more conservative manner. Assume the development project is only willing to absorb the amortized cost of the component used in the project. That is, if the project is using only the requirements from the acquired component, we can argue that requirements cost is the only cost to be amortized. In that case, the model becomes:

$$C = (b + \frac{a(1-b)}{n} - 1)R + 1 \qquad (6)$$

where,

$a(1-b)$ represents the requirements acquisition cost. We cannot ignore the cost of maintaining the library of reused components. Let the cost of library maintenance be allocated as a fraction of the component development cost. Incorporating maintenance into Equation (5) we find:

$$C = [b + \frac{a(1-b+d)}{n} - 1]R = 1 \qquad (7)$$

where,

d is the cost fraction added to the component acquisition cost to account for reuse library maintenance. The maintenance fraction value is a function of the size and use of the maintenance library. The maintenance value is also amortized over the number of component uses.

Acquisition Amortization

The reusable component amortization is a function of the number of applications of each component. A large number of reuses n reduces the magnitude of the amortization factor a/n in each of Equations (5) - (7). The reuse cost coefficient

$$b + \frac{a}{n} - 1 \qquad (8)$$

must be negative in order to provide a cost improvement in Equation (5). Or, in other words, if the coefficient is $b+a/n<1$, the relative software development cost C for the project is less than 1.0.

	Relative Cost of Developing Reuse Component (*a*)				
Relative Reuse Cost (*b*)	**1.00**	**1.25**	**1.50**	**1.75**	**2.00**
Requirements (0.93)	15	18	22	25	29
Design (0.62)	3	4	4	5	6
Code (0.39)	2	3	3	3	4
Validated Code (0.27)	2	2	3	3	3

Table 3: *Minimum Reuse Number (n_0) Versus Component Type (b) and Acquisition Cost (a)*

The minimum number of reuse applications can be derived from Equation (8) by setting the coefficient to unity and solving for n. The resulting equation shown in Equation (9) represents the number of uses required to cover the reusable component cost. The threshold reuse number (n_0) is:

$$n_0 = \text{ceiling}\ (\frac{a}{1-b}) \qquad (9)$$

rounded up to the nearest unit. *Ceiling(arg)* is defined as the smallest integer greater than, or equal to, *arg*. The information in Table 3 demonstrates the threshold, or minimum number of reuse applications.

"The models developed in this effort and the results achieved here are independent of software estimating tools or models. This information can be tailored or related to any software cost estimation model."

The threshold reuse number values shown in Table 3 represent the break-even values for reusable component development. The number of reuse applications must be greater than, or equal to, the numbers shown to have a positive impact on the projects using the components.

COTS Cost Model

COTS software is a special application of software reuse. There are several assumptions we must make before specifying the COTS software cost model. The best way to visualize COTS software is as a shrink-wrapped product. This basically means that the software includes the following:

- Contains only validated source code.
- Is purchased and not internally developed or modified.
- Has no library costs associated with the product.
- Conforms to the black-box definition.
- Requires no product maintenance.
- Requires no version upgrades.

The reuse fraction R is approximated by estimating the source code size for an internally developed product that is functionally equivalent to the COTS software. The ratio R is defined as:

$$R = \frac{S_{COTS}}{S_e + S_{COTS}} \qquad (10)$$

where,

S_{COTS} is the estimated size of an internally developed COTS replacement, and S_e is the effective size of the new, modified, and lifted source code S_o as shown in Figure 2.

Externally developed components (COTS) are simpler to analyze because the development costs are outside the project development environment. Amortization and maintenance costs are still relevant to the economic analysis. The economic cost model for COTS software (validated code) becomes:

$$C = (\frac{F(1+d)}{n} - 0.73)R + 1 \qquad (11)$$

where,

F is the COTS acquisition cost relative to the cost of non-reusable development from scratch. The lower cost limit for free COTS components is approximately 27 percent due to regression testing and software validation. No consideration has been given in Equation (11) to the costs associated with component evaluation and selection, nor has any consideration been allowed for developing expertise in the components' external interface or function.

We can graphically illustrate the relative software costs associated with using COTS software. Let us consider the following example:

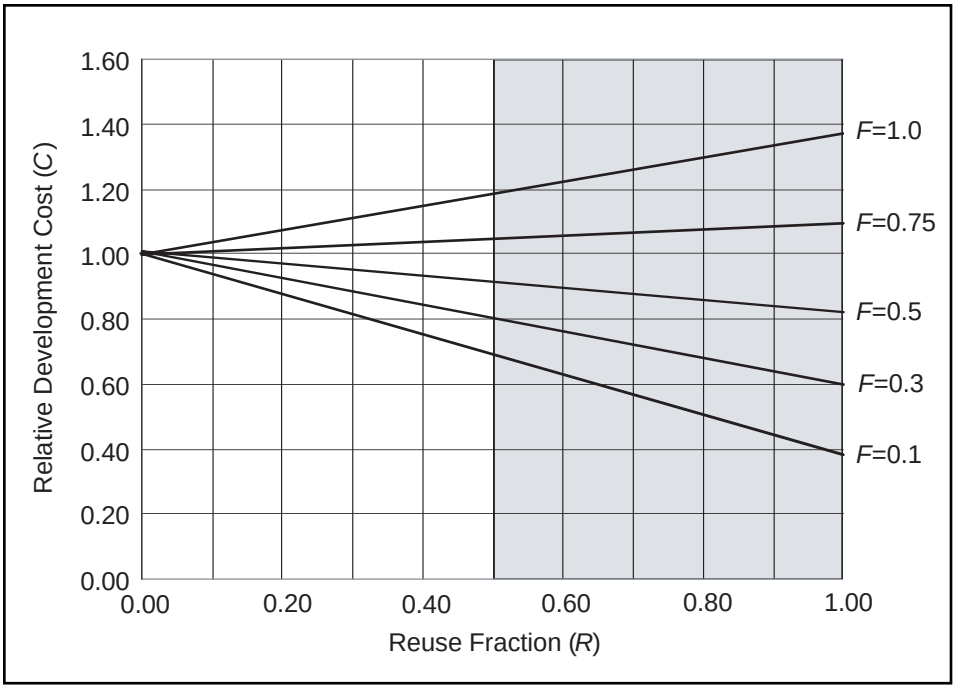


Figure 4: *Relative Development Cost Versus Reuse Fraction By Relative COTS Purchase Cost*

- F is in the practical range $0.1 \le F \le 1$; that is, F is limited to the cost of developing the COTS component(s) from scratch.
- Component cost is to be amortized over one (1) application, or $n=1$.
- Maintenance over the useful life of the component(s) is 10 percent, or $d=0.1$.

The relative product software cost C relative to the cost of all new source code calculated from Equation (11) for this example is plotted in Figure 4. The maximum relative acquisition cost F in this model under these conditions to break even is approximately 65 percent of the cost to develop the COTS product from scratch.

If we assume the reusable component is free ($F=0$) and a practical maximum reuse fraction ($R=0.5$), the economic model in Equation (11) shows the relative development cost is approximately 64 percent. The ideal relative development cost with a reuse fraction for $R=0.5$ is 50 percent of the cost of developing the product without reusable components. The economic model prediction is realistically higher than the ideal condition.

Summary and Conclusions

The intent of this effort produced a simplified economic model that provides a realistic prediction of software product development costs in an environment containing reused software components. The reuse definition used in this analysis includes both COTS software and internal software components developed for reuse. The models are developed at two levels. The first level, a truly first-order model, relates relative software product development costs to the fraction of the product to be implemented by reusable components and the reusable component sophistication (requirements, design, etc.). The second level incorporates the significant costs associated with the development, or acquisition of reusable components.

The models developed in this effort and the results achieved here are independent of software estimating tools or models. This information can be tailored or related to any software cost estimation model.

The economic model does not attempt to account for all costs associated with software reuse. The reusable component function and interface complexity issues are ignored here, but are vital estimate elements in practice. There are several cost factors not included because of the difficulty in establishing numeric values for these factors in a broad general sense. These factors, listed in the introduction, should not be ignored in the real application of these models. The factors are major considerations in most projects.◆

References

1. Gaffney Jr., John E., and Thomas A. Durek. "Software Reuse – Key to Enhanced Productivity; Some Quantitative Models." Vers. 1.0. SPC-TR-88-015. Herndon, VA: Software Productivity Consortium, Apr. 1988.
2. Gaffney Jr., 2-1.
3. Software Engineering, Inc. Sage User's Guide. Brigham City, UT: Software Engineering, Inc., May 2001.
4. Boehm, B.W. Software Engineering Economics. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.
5. REVIC Users Group. REVIC Software Cost Estimating Model User's Manual. Vers. 9.0. Arlington, VA: Air Force Cost Center, 1991.

About the Author

Randall W. Jensen, Ph.D., is a consultant for the Software Technology Support Center, Hill Air Force Base, Utah, with more than 40 years of practical experience as a computer professional in hardware and software development. For the past 30 years, he has actively engaged in software engineering methods, tools, quality software management methods, software schedule and cost estimation, and management metrics. He retired as chief scientist of the Software Engineering Division of Hughes Aircraft Company's Ground Systems Group, and was responsible for research in software engineering methods and management. Jensen founded Software Engineering, Inc., a software management-consulting firm in 1980. He developed the model that underlies the Sage and the Galorath Associates, Inc.'s Software Evaluation and Estimation of Resources – Software Estimating Model [SEER-SEM] software cost and schedule estimating systems. Jensen received the International Society of Parametric Analysts Freiman Award for Outstanding Contributions to Paramet-ric Estimating in 1984. He has published several computer-related texts, including "Software Engineering," and numerous software and hardware analysis papers. He has a Bachelor of Science in electrical engineering, a Master of Science in electrical engineering, and a doctorate in electrical engineering from Utah State University.

Software Technology Support Center
6022 Fir AVE, BLDG 1238
Hill AFB, UT 84056-5820
Phone: (801) 775-5742
Fax: (801) 777-8069
E-mail: randall.jensen@hill.af.mil

Estimating and Managing Project Scope for Maintenance and Reuse Projects

William Roetzheim
Cost Xpert Group, Inc.

Estimating project scope is considered by many to be the most difficult part of software estimation. Parametric models have been shown to give accurate estimates of cost and duration given accurate inputs of the project scope, but how do you input scope early in the life cycle when the requirements are still vaguely understood? How can scope be estimated, quantified, and documented in a manner that is understandable to management, end users, and estimating tools? This article focuses on scope estimates for maintenance and reuse work, including bug fixes (corrective maintenance); modifications to support changes in the operating system, database management system, compiler, or other aspect of the operating environment (adaptive maintenance); and modifications of existing functionality to improve that functionality (perfective maintenance). Reuse includes any case where you are modifying an existing code base to support enhanced functionality, and includes cases where an existing application is translated to a new language. The effort estimates cover code fixes and enhancements, regression and other testing of those fixes, updates to documentation, and management of those efforts. It does not include requirement/usability efforts or deployment efforts.

At a high level, maintenance projects consist of three types of work:

1. Maintaining an existing, functioning application.
2. Modifying existing code to support changing requirements.
3. Adding new functionality to an existing application.

A team doing a new build for an existing application would only be concerned with item Nos. 2 and 3. A team keeping an existing code base functioning would only do item No. 1, and possibly item No. 2 depending on how new builds are handled. A project manager may be responsible for both areas and might need to estimate the effort required for all three. This article will deal with each individually.

Maintaining Existing Code

Maintenance as we are defining it consists of three types of activities [1]:

- **Corrective Maintenance.** Fixing bugs in the code and documentation. Bugs are areas where the code does not operate in accordance with the requirements used when it was built.
- **Adaptive Maintenance.** Modifying the application to continue functioning after installation of an upgrade to the underlying virtual machine (database management system, operating system, etc.).
- **Perfective Maintenance.** Correcting serious flaws in the way it achieves requirements (e.g., performance problems).

Maintenance effort is a function of the development effort spent on the original project. (Figure 1 shows an example of a commercial tool reuse parameter screen). The larger the original project in terms of effort, the more staff must be assigned to maintain the application. A second factor is Annual Change Traffic (ACT), or the percent of the code base that will be touched each year as a result of maintenance work. Numbers for ACT between 3 percent and 20 percent are reasonable, with 12 percent to 15 percent being fairly typical. The Nominal Annual Maintenance effort while the application is in a maintenance steady state will equal

ACT x PM

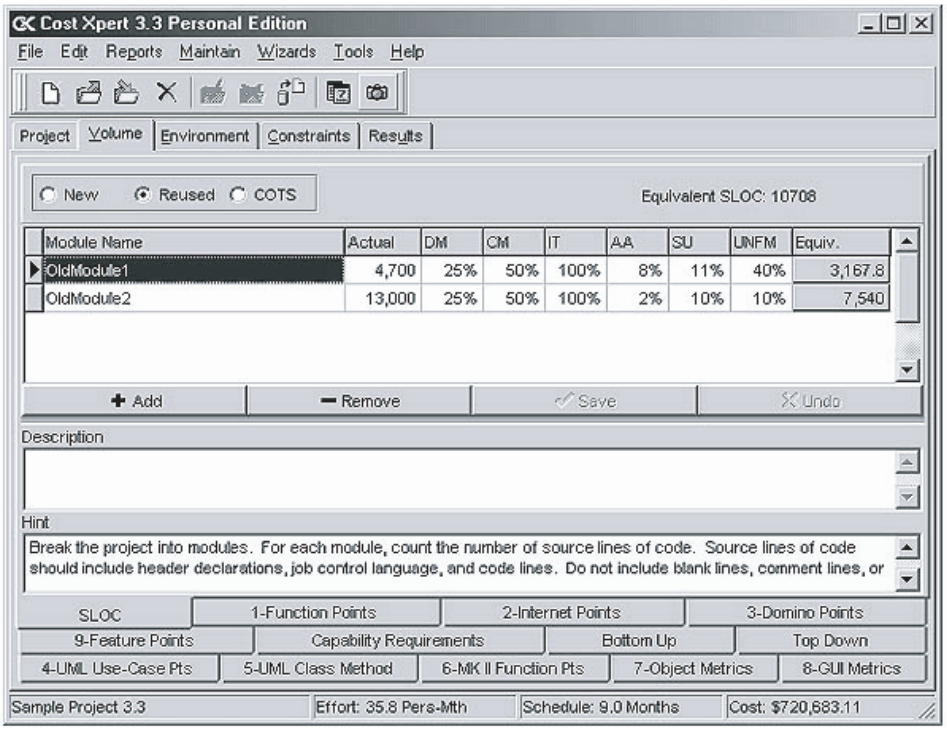
where,

PM is the original person months of development effort.

Maintenance steady state is typically achieved in year four following software delivery. Between software delivery and steady state, the maintenance effort follows a Rayleigh[1] curve starting with 1.5 x ACT x PM in year one and dropping down to the steady state value. Software maintained beyond nine years typically sees maintenance costs begin to climb again. This is due to a combination of increasingly fragile code and an increasing distance between the technology used for development and the current state of the art.

If you know the effort spent on the original development, the above equations may be used as shown. If you do not know the original development effort, you must first estimate that effort, normally using a commercial estimation tool. This will require that you count the physical lines of code (or function points), and then either make educated guesses at values for environmental variables (team capability and so

Figure 1: *Example of Commercial Tool Reuse Parameter Screen*



on) or use the tool's default values.

In some cases, strategic considerations will require that programmers with knowledge of the product be kept available and current to facilitate future planned or potential modifications of the code. These developers would then be available to make required short fixes to the code as well. In this case, factors outside the scope of this article would dictate the number of developers that must be kept available and on the maintenance staff.

Modifying Existing Code

The basis of code modification is very simple: Code already exists that may be utilized in any given project. You begin by actually counting the values measured in the existing application(s). For example, if using lines of code (LOC), employ a code counting utility to physically count the lines of code in the existing program modules (or use the values from your configuration management system with an adjustment to remove the impact of blank lines and comments). If using Function Points, count the existing reports, screens, tables, and so on.

Calculating the Equivalent Volume

Our goal is to convert from the known value for the volume of reusable code to an equivalent volume of new code. Think about it this way:

- If we have 100 function points worth of reusable code but the reusable code is worth nothing to us, then no effort will be saved; the equivalent amount of new code is 100 function points.
- If we have 100 function points worth of reusable code and we can reuse it without any changes, retesting, or integration whatsoever, then using the code is a *freebie* from a developmental perspective. The equivalent amount of new code is 0 function points.
- If we have 100 function points worth of reusable code and this saves us half the effort relative to new code, then the equivalent amount of new code is 50 function points.

We convert from reused volume values to equivalent new volume values by looking at six factors: Percent Design Modification (PDM), Percent Code Modification (PCM), Percent Integration and Testing (PI&T), Assessment and Assimilation (AA), Software Understanding (SU), and Unfamiliarity (UNFM) with software.

Percent Design Modification

The PDM measures how much design effort the reused code will require. Basically, a low percent value indicates high code reuse, whereas a high percent value indicates low code reuse and increases the requirement to develop new code as follows:

- A value of 0 percent says that the reused code is perfectly designed for the new application and no design time will be required at all.
- A value of 100 percent says that the design is totally wrong and the existing design will not save any time at all.
- A value of 50 percent says that the design will require some changes and that the effort involved in making these changes is 50 percent of the effort of doing the design from scratch.

For typical software reuse, the PDM will vary from 10 percent to 25 percent.

Percent Code Modification

The PCM measures how much we will need to change the physical source code as follows:

- A value of 0 percent says that the reused code is perfect for the new application, and the source code can be used without change.
- If the reused code was developed in a different language and you need to port the code to your current language, the value would be 100 percent (ignoring any potential automated translation using automatic translation tools).
- Numbers in between imply varying amounts of code reuse.

The PCM should always be at or higher than the PDM. As a rule of thumb, we have found the PCM is often twice the PDM.

Percent Integration and Testing

The PI&T measures how much integration and testing effort the reused code will require as follows:

- A value of 0 percent would mean that you do not anticipate any integration or integration test effort at all.
- A value of 100 percent says that you plan to spend just as much time integrating and testing the code that you would if it was developed new as part of this project.
- Numbers in between simply refer to differing degrees of integration and testing effort relative to new development.

The PI&T should always be at or higher than the PCM. It is recommended that you set the PI&T to at least twice the PCM.

It is not unusual for this factor to be 100 percent, especially for mission-critical systems where the risk of failure is significant. For commercial off-the-shelf components (purchased libraries) where the PDM and PCM are often zero, it is not unusual to see a number of 50 percent here to allow for the integration effort and time spent testing the application with the commercial component.

Finally, after measuring your existing code's volume and estimating your PDM, PCM, and PI&T, calculate the Adaptation Adjustment Factor (AAF) where AAF is the:

$$AAF = .4DM + .3CM + .3\ I\&T \qquad (1)$$

where,

this equation comes from [2][2].

Suppose that we need to implement a new e-commerce system consisting of 15,000 source lines of code (SLOC). Let us ignore environmental adjustments for the moment.

Assessment and Assimilation

AA indicates how much time and effort will be involved in testing, evaluating, and documenting the screens and other parts of the program to see what can be reused. Values range from 0 percent to 8 percent.

Software Understanding

SU estimates how difficult it will be to understand the code once you are modifying it, and how conducive the software is to being understood. Is the code well structured? Is there good correlation between the program and application? Is the code well commented? The range of possible values is a numeric entry between 10 percent and 50 percent, default 30 percent.

Unfamiliarity With Software

UNFM with software indicates how much your team has worked with this reusable code before. Is this their first exposure to it, or is it very familiar? The range of possible values is between 0 percent and 100 percent, default 40 percent.

Using the Six Factors

Three of the factors – AA, SU, and UNFM with software – add a form of tax to software reuse, compensating for the overhead effort associated with reusing code. For projects where the amount of reuse is small (AAF is less than or equal to 50 percent), the following formula applies with adjustments per the above factors:

$$ESLOC = ASLOC \times [AA + AAF(1 + 2 \times SU \times UNFM)]$$

where,

ESLOC is the equivalent SLOC and

ASLOC is the actual SLOC.

Before discussing how this equation is used to determine the reuse effort, let us take a step back to discuss a simple equation to determine effort. If you are aware of the number of thousand SLOC (KSLOC) your developers must write, and you know the effort required per KSLOC, then you could multiply these two numbers together to arrive at the person months of effort required for your project.

$$\text{Effort} = \text{Productivity} \times \text{KSLOC}$$

where,

KSLOC represents a measure of program scope.

Table 1 shows some common values that Cost Xpert researchers have found for these linear productivity factors. The COCOMO II value comes from research by Barry Boehm at the University of Southern California (USC). The values for embedded, e-commerce, and Web development come from Cost Xpert research working with a variety of organizations, including IBM and Marotz.

You also must consider that researchers have found that productivity varies with project size. In fact, large projects are significantly less productive than small projects. The probable causes are a combination of increased coordination and communication time, plus more rework required due to misunderstandings.

This productivity decrease with increasing project size is factored in by raising the number of KSLOC/thousand software LOC to a power greater than 1.0. This exponential factor then penalizes large projects for decreased efficiency. Table 2 shows some typical size penalty factors for various project types. Again, the COCOMO II value comes from work by Barry Boehm at USC; values for embedded, e-commerce, and Web come from work by Cost Xpert Group and our customers. Note that because the size factor is an exponential factor, rather than linear, it does not change with project size, but rather changes in impact on the end result with project size.

As seen in the tables, the productivity and penalty constants vary by project and organization. Let us take an example involving 15,000 reused SLOC. Using the following formula, as well as the productivity and size penalty factors for e-commerce development, the predicted effort will be:

Project Type	Linear Productivity Factor (PM/KSLOC)
COCOMO II Default	3.13
Embedded Development	3.60
E-Commerce Development	3.08
Web Development	2.51

Table 1: *Common Values for Linear Productivity Factor*

$$\begin{aligned}\text{Effort} &= \text{Productivity} \times \text{KSLOC}^{\text{Penalty}} \\ &= 3.08 \times 15^{1.030} \\ &= 3.08 \times 16.27 \\ &= 50 \text{ Person Months}\end{aligned}$$

Suppose we found that we could get by with 10 percent design modifications, 20 percent code modifications, and 40 percent integration and test effort. AAF would then be calculated as:

$$\text{AAF} = (0.4 \times 0.1) + (0.3 \times 0.2) + (0.3 \times 0.4) = 0.22$$

Because AAF is less than or equal to 50 percent we can use the formula just presented. Now, suppose that AA was 4 percent, SU was 30 percent, and UNFM was 40 percent. The equivalent source lines of code (ESLOC) would now be:

$$\text{ESLOC} = 15{,}000 \left[0.04 + 0.22(1 + 2 \times 0.3 \times 0.4)\right] = 4{,}692$$

Using our earlier assumptions, the effort required to build this software would be:

$$\begin{aligned}\text{Effort} &= \text{Productivity} \times \text{ESLOC}^{\text{Penalty}} \\ &= 3.08 \times 4.692^{1.030} \\ &= 3.08 \times 4.915 \\ &= 15.14 \text{ Person Months}\end{aligned}$$

The formula when reuse is low and AAF is less than 50 percent changes. The formula in this situation is:

$$\text{ESLOC} = \text{ASLOC} \times \left[\text{AA} + \text{AAF} + (\text{SU} \times \text{UNFM})\right]$$

Let us work through our same example of 15,000 lines of reused code, but let us now suppose that the design modification is 50 percent, the code modification 100 percent, the integration and test are 100 percent, and the correct values for AA, SU, and UNFM are 8 percent, 50 percent, and 100 percent respectively.

AAF is now calculated as:

$$\text{AAF} = (0.4 \times 0.5) + (0.3 \times 1.0) + (0.3 \times 1.0) = 0.8$$

Because AAF is over 50 percent, we use the formula as follows:

$$\begin{aligned}\text{ESLOC} &= 15{,}000 \times [0.08 + 0.8 + 0.5 \times 1.0] \\ &= 15{,}000 \times 1.38 \\ &= 20{,}700\end{aligned}$$

Effort is now calculated as:

$$\begin{aligned}\text{Effort} &= \text{Productivity} \times \text{ESLOC}^{\text{Penalty}} \\ &= 3.08 \times 20.7^{1.030} \\ &= 3.08 \times 22.67 \\ &= 69.82 \text{ Person Months}\end{aligned}$$

In this case, as seen by comparing the person months in the first example of this article with the person months in the final example, reusing those 15,000 LOC actually takes 19.82 person months more effort than writing the same code from scratch! In fact, this phenomenon is even more pronounced than shown in the preceding example. If you need 15,000 lines of new functionality, you will seldom find a reusable block of code that exactly matches the functionality you are looking for. More often, the reused code will be significantly larger than the new code because it will do many functions that you are not interested in.

Perhaps you will be reusing a piece of code that is 25,000 LOC in size, all to get at those 15,000 lines of code worth of functionality that you care about. Well, the entire 25,000 LOC will typically need to be assessed, understood, and tested to some degree. The end result is that in general, you will find that somewhere between 15 percent and 30 percent design change is the crossing point beyond which you are typically better off rewriting the code from scratch. The correct value in

Table 2: *Penalty Factors for Various Project Types*

Project Type	Exponential Size Penalty Factor
COCOMO II Default	1.072
Embedded Development	1.111
E-Commerce Development	1.030
Web Development	1.030

Coming Events

December 2-3
The 6th IEEE Workshop on Mobile Computing Systems and Applications
Lake Windermere, United Kingdom
http://wmcsa2004.lancs.ac.uk/

December 2-4
International Conference on Intelligent Technologies (InTech) '04
Houston, TX
http://csc.csudh.edu/intech04/index.htm

December 4-8
IEEE/ACM International Symposium on Microarchitecture
Portland, OR
www.microarch.org/micro37/

December 6-9
Interservice Industry Training, Simulation, and Education Conference
Orlando, FL
www.iitsec.org

January 6-9, 2005
Internet, Processing, Systems, and Interdisciplinary Research (IPSI) 2005
Oahu, HI
www.internetconferences.net/industrie/hawaii2005.html

January 9-12
International Conference on Intelligent User Interfaces
San Diego, CA
www.iuiconf.org/

January 31-February 3
16th Annual Government Technology Conference
Austin, TX
www.govtech.net/gtc/?pg=conference&confid=182

April 18-21
2005 Systems and Software Technology Conference
Salt Lake City, UT
www.stc-online.org

this range will depend largely on how well matched the reused code is to your requirements and the quality of that code and documentation.

If you are doing an ongoing series of maintenance builds with a large, relatively stable application, there are some tricks to simplify your planning. Create a spreadsheet containing all of the modules and for each module, the LOC in that module. Set percent design mode, code mode, and so on to zero for each module in the spreadsheet. It is also useful in the spreadsheet to include an area where you identify the dependent relationships between modules (this can sometimes be done using a tool like Microsoft Project, where you treat each module as a task in the dependency diagram). Save this as your master template for planning a new build.

When you are planning a build, analyze each requirement for change to identify the modules that must be modified and fill in the appropriate value for design modification, code modification, etc. Then, look at the modules that are dependent on these modules and put in an appropriate value for IP&T for those dependent modules. You can then quickly calculate the resultant equivalent scope and use this to calculate a schedule and the effort required. For the next build, go back to the template you started with and repeat the process. Some commercial estimating tools support this approach, as well.

Adding New Functionality

Finally, when preparing a new software build there are normally some areas where completely new functionality is added to the system. This functionality is defined and estimated as new development using the standard approaches suitable for estimating new software development.

Conclusions

This article presents quantitative approaches to estimating scope and effort for maintenance, enhancement, and reuse projects. Following these techniques will produce reasonable and justifiable estimates and budgets for maintenance projects, and help with build release planning.◆

References

1. Cost Xpert Group. Cost Xpert Vers. 3.3 User Manual. Rancho San Diego, CA: Cost Xpert Group, Inc., 19 Nov. 2003.
2. Boehm, Barry W., et al. Software Cost Estimation With COCOMO II. 1st ed. Prentice Hall PTR, 15 Jan. 2000.

Notes

1. A Rayleigh curve yields a good approximation to the actual labor curves on software projects.
2. The work in this paper is heavily dependent on work by Barry W. Boehm and others, as documented in this latest book [2].

About the Author

William Roetzheim has 25 years experience in the software industry and is the author of 15 software related books and over 100 technical articles. He is the founder of the Cost Xpert Group, Inc., a Jamul-based organization specializing in software cost estimation tools, training, processes, and consulting.

Cost Xpert Group
2990 Jamacha RD STE 250
Rancho San Diego, CA 92019
E-mail: william@costxpert.com

Web Sites

Reusable Software Research Group

www.cse.ohio-state.edu/rsrg

The Reusable Software Research Group (RSRG) began at Ohio State University in the mid 1980's and is currently active at Ohio State, at Clemson University, and at Virginia Tech. The RSRG deals with the disciplined engineering of component-based software systems and the software components (aka reusable software components) from which they can be built. The RSRG's work involves a framework for component-based software engineering, a research language, and a component-design discipline called RESOLVE. The group is concerned with creating software that is at once reusable, efficient, verifiable, and comprehensible.

RESOLVE

http://people.cs.vt.edu/~edwards/resolve

RESOLVE is a comprehensive and robust framework, discipline, and language for the construction of highly reusable component-based software. The RESOLVE Web site contains links to online readings and an annotated bibliography for those interested in this work.

Using Java for Reusable Embedded Real-Time Component Libraries

Dr. Kelvin Nilsen
Aonix

Java as a high-level programming language provides great support for a wide variety of networked devices and embedded systems. When used in a military context, Java promises to reduce development and maintenance costs significantly, while increasing reliability, flexibility, and functionality of embedded systems. The secret in Java's success lies in its ability to provide real-time mission-critical response. This article discusses the characteristics of mature Java technologies that are able to meet these important defense criteria.

Originally designed as a language to support "advanced software for a wide variety of networked devices and embedded systems" [1], the Java programming language has much to offer the community of embedded defense system developers. In this context, Java is considered a high-level, general-purpose programming language rather than a special-purpose Web development tool. Java offers many of the same benefits as Ada, while appealing to a much broader audience of developers. The breadth of interest in Java has led to a large third-party market for Java development tools, reusable component libraries, training resources, and consulting services.

Java borrows the familiar syntax of C and C++. Like C++, Java is object-oriented, but it is much simpler than C++ because Java's designers chose not to support compilation of legacy C and C++ code. Due to its simplicity, more programmers are able to master the language. With that mastery, they are more productive and less likely to introduce errors resulting from misunderstanding the programming language.

The Java write-compile-debug cycle is faster than with traditional languages because Java supports both interpreted and just-in-time (JIT) compiled implementations. During development and rapid prototyping, developers save time by using the interpreter. This avoids the time typically required to recompile and relink object files.

Java application software is portable because the Java specification carefully defines a machine-independent intermediate byte-code representation and a robust collection of standard libraries. Byte-code class files can be transferred between heterogeneous network nodes and interpreted or compiled to native code on demand by the local Java run-time environment. The benefits of portability are four-fold:

1. Software engineers can develop and test their embedded software on fast PC workstations with large amounts of memory, and then deploy on smaller, less powerful embedded targets.
2. As embedded products evolve, it is often necessary to port their code from one processor and operating system to another.
3. Cross compiling is no longer necessary. The same executable byte code runs on Power PC, Pentium, MIPS, XScale, and others. This simplifies configuration management.
4. The ability to distribute portable binary software components lays the foundation for a reusable software component industry.

Certain features in Java's run-time environment help to improve software reliability. For example, automatic garbage collection, which describes the process of identifying all objects that are no longer being used by the application and reclaiming their memory, has been shown to reduce the total development effort for a complex system by approximately 40 percent [2]. Garbage collection eliminates dangling pointers and greatly reduces the effort required by developers to prevent memory leaks.

A high percentage of the Computer Emergency Response Team advisories issued every year are a direct result of buffer overflows in system software. Java automatically checks array subscripts to make sure code does not accidentally or maliciously reach beyond the ends of arrays, thereby eliminating this frequently exploited loophole.

The Java compiler and class loader enforce type checking much more strongly than C and C++. This means programmers cannot accidentally or maliciously misuse the bits of a particular variable to masquerade as an unintended value.

Finally as part of the interface definition for components, Java component developers can require that exceptions thrown by their components be caught within the surrounding context. In lower level languages, uncaught exceptions often lead to unpredictable behavior.

Another very useful Java feature is the ability to dynamically load software components into a running Java virtual machine environment. New software downloads serve to patch errors, accommodate evolving communication protocols, and add new capabilities to an existing embedded system. Special security checking is enforced when dynamic libraries are installed to ensure they do not compromise the integrity of the running system.

Though Java programs may be interpreted, it is much more common for Java byte codes to be translated to the target machine language before execution. For many input/output-intensive applications, compiled Java runs as fast as C++. For compute-intensive applications, Java tends to run at one-third to one-half the speed of comparable C code.

Java Within Real-Time Systems

Although Java's initial design was targeted to embedded devices, its first public distributions did not support reliable real-time operation. Several specific issues are identified here and discussed in greater detail in the reference material [3, 4].

Automatic Garbage Collection

Though automatic garbage collection greatly reduces the effort required by software developers to implement reliable and efficient dynamic memory management, typical implementations of automatic garbage collection are incompatible with real-time requirements. In most virtual machine environments, the garbage collector will occasionally put all the application threads to sleep during certain uninterruptible operations while it analyzes the relationships between objects within the heap to determine those no longer in use. The durations of these garbage collection pauses are difficult to predict, and typically vary from half a second to tens of seconds. These problems can be addressed by using a virtual machine that provides real-time garbage collection as described in the following.

Priority Inversion

To guarantee that real-time tasks meet all of their deadlines, real-time developers carefully analyze the resource requirements of each

task and set their priorities according to accepted practices of scheduling theory [5]. Thread priorities are used as a mechanism to implement compliance with deadlines. Unfortunately, many non-real-time operating systems and most Java virtual machine implementations view priorities as heuristic suggestions. This compromises real-time behavior whenever the priorities of certain threads are temporarily boosted in the interest of providing *fair* access to the CPU or to improve overall system throughput. Another problem occurs when low-priority tasks lock resources that are required by high-priority tasks. In all of these cases, the real-time engineer describes the problem as *priority inversion*.

It is important to deploy real-time Java components on virtual machines that honor strict priorities, preventing the operating system from automatically boosting or aging thread priorities, and that build priority inheritance or some other priority inversion avoidance mechanism into the implementation of synchronization locks. Priority inheritance, for example, elevates the priority of a low-priority thread that owns a lock being requested by a high-priority thread so that the low-priority thread can get its work done and release the lock.

Timing Services

Standard Java timing services do not provide the exacting precision required by real-time programmers. Applications that use the Java sleep() service to control a periodic task will drift off schedule because each invocation of sleep() is delayed within its period by the time required to do the periodic computation, and because the duration of each requested sleep() operation only approximates the desired delay time. Also, if a computer user changes the operating system's notion of time while a real-time Java program is running, the real-time threads will become confused because they assume the system clock is an accurate, monotonically increasing time reference. Java virtual machines designed for real-time operation typically provide high-precision and drift-free real-time timers that complement the standard timing utilities.

Low-Level Control

As a modern, high-level programming language, Java's design intentionally precludes developers from directly accessing hardware and device drivers. The ideal is that hardware device drivers should be abstracted by the underlying operating system. However, if Java is up to the task many software engineers would rather do that development in Java than in assembly language or C. Most real-time Java implementations provide services to allow real-time Java components to store and fetch values from input/output ports and memory-mapped devices.

Hard Real-Time Tradeoffs

Developers of hard real-time systems tend to make different tradeoffs than soft real-time developers. Hard real-time software tends to be relatively small, simple, and static. Often, economic considerations demand very high performance and very small footprint of the hard real-time layers of a complex system. To meet these requirements, hard real-time developers generally recognize they must work harder to deliver functionality that could be realized with much less effort if there were no timing constraints, or if all of the timing constraints were soft real-time. Work is under way to define special hard real-time variants of the Java language [6, 7, 8, 9]. One noteworthy difference is that the hard real-time variants generally do not rely on any form of automatic garbage collection.

Real-Time Garbage Collection

One of the most difficult challenges of real-time development with Java is managing the interaction between application code and automatic garbage collection. For reliable operation, there are a number of characteristics that must be satisfied by the garbage collection subsystem. These are described in the following sections.

Preemptive

Typical real-time Java applications are divided into multiple threads, some allocate memory and others manipulate data in previously allocated objects. Both classes of threads may have real-time constraints. Threads that do not allocate memory may have tighter deadlines and run at higher priorities than threads that do allocate memory. Garbage collection generally runs at a priority level between these two classes of priorities. Whenever a higher priority thread needs to run, it must be possible to preempt garbage collection. In some non-real-time virtual machines, once garbage collection begins, it cannot be preempted until it has executed.

Incremental

To assure that garbage collection makes appropriate forward progress, it is necessary to divide the garbage collection effort into many small work increments. Whenever garbage collection is preempted, it must resume with the next increment of work after the preempting task relinquishes control. Real-time garbage collectors avoid the need to restart operations when garbage collection is preempted.

Accurate

We use the term *accurate* to describe a garbage collector that always knows whether a particular memory cell holds a reference (pointer) or holds, for example, numerical representations of integers and floating-point values. In contrast, *conservative* garbage collectors assume memory cells contain pointers whenever there is any uncertainty about the cell's contents. If, interpreted as a pointer, there is an object that would be directly referenced by this pointer, then that object is conservatively treated as live. Because conservative garbage collectors cannot promise to reclaim all dead memory, they are less reliable for long-running mission-critical applications.

Defragmenting

Over the history of a long-running application, it is possible for the pool of free memory to become fragmented. While a fragmented allocation pool may have an abundance of available memory, the free memory is divided into a large number of very small segments, which prevents the system from reliably allocating large objects. It also complicates the allocation of smaller segments because it becomes increasingly important to efficiently pack newly allocated objects into the available free memory segments (to reduce further fragmentation). In general, a real-time virtual machine intended to support reliable, long-running, mission-critical applications must provide some mechanism for defragmentation of the free pool.

Paced

It is not enough to just preempt garbage collection. In large and complex systems, certain activities depend on an ability to allocate new memory to fulfill their real-time-constrained responsibilities. If the memory pool becomes depleted, the real-time tasks that need to allocate memory will necessarily become blocked while garbage collection executes. To prevent this priority inversion from occurring, a real-time virtual machine must pace garbage collection against the rate of memory allocation.

Ideally, the system automatically dedicates to garbage collection activities enough CPU time to recycle dead memory as quickly as the application is allocating memory. However, it does so without dedicating any CPU time that has already been set aside for execution of the real-time application threads, and without dedicating so much CPU time that it completes way ahead of schedule. In a soft or firm real-time system, heuristics are applied to approximate this ideal. The driving considerations are (1) to prevent out-of-memory conditions from

stalling execution of real-time threads, and (2) to maximize garbage collection efficiency by delaying it as long as possible so that each fixed-cost collection reclaims the largest possible amount of dead memory.

Pacing Garbage Collection

Think of garbage collection as a servant to all of the application threads that regularly allocate memory. Because the total garbage collection effort consists of many incremental steps that ultimately benefit all threads that allocate memory, the priority assigned to garbage collection activities must be greater than or equal to that of the highest priority application thread that performs memory allocation.

A pacing agent is the software component responsible for allocating CPU time to the garbage collection effort. In configuring the pacing agent, it is important to identify the maximum priority of threads that allocate memory, the minimum priority assigned to threads having real-time execution constraints, the shortest deadline corresponding to real-time allocating threads, and the percentages of CPU time to be reserved for execution of real-time allocating and non-allocating application threads respectively.

Also, the pacing agent monitors the application to discover behavioral trends, including the rate of memory allocation and the amount of live memory retained following completion of each garbage collection pass. The pacing agent combines all of this information into a coherent approximation of the application's resource requirements. The pacing agent uses this approximation to guide its allocation of CPU time increments to the garbage collection effort.

Among the heuristics applied by the pacing agent are the following:

1. For a given phase of execution, assume future memory allocation behavior resembles previous allocation behavior. An application programming interface service allows the application to inform the pacing agent each time it changes phases. Within a phase, we assume that allocation rates are constant and that retained live-memory is linear in time. This model approximates typical phases such as initialization (during which time large data structures are constructed), steady state execution (during which each new allocation is matched by release of a previously allocated object of similar size), and termination (during which time large data structures may be disassembled).
2. The pacing agent treats garbage collection as a real-time activity with priority at least as high as that of the highest allocating real-time thread. If, however, rate monotonic analysis concludes that there are insufficient CPU resources to guarantee that garbage collection will stay on pace with allocation, the pacing agent endeavors to steal additional CPU cycles for garbage collection at the priority immediately below the lowest priority real-time allocating thread. The pacing agent assures that garbage collection never consumes more CPU time at real-time priorities than would be available according to the rules of rate monotonic analysis [5].
3. To not compromise deadline compliance of real-time allocating threads, the pacing agent takes special care to ensure that its triggering of real-time increments of garbage collection work is periodic with a period no longer than the shortest deadline of the allocating real-time threads. Otherwise, it might end up with the higher priority garbage collection thread having a longer deadline than the lower priority allocating real-time threads, and this would compromise the results of rate monotonic analysis.

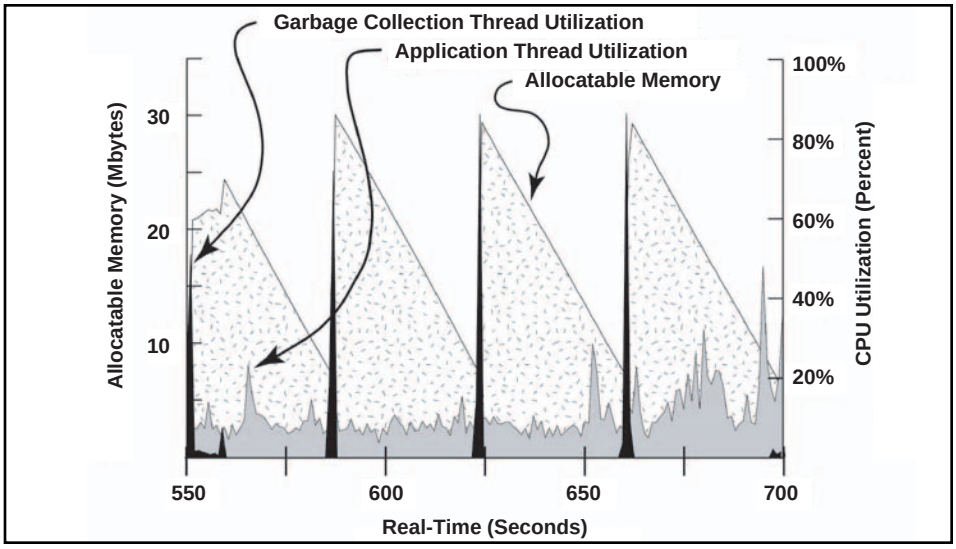
In Figure 1, the amount of allocatable memory is represented by the hashed saw tooth shape, measured according to the scale on the left side of the chart. The amount of CPU time consumed by the simulated air traffic control system's real-time Java application threads is shown in gray. The percentage of CPU time dedicated to real-time garbage collection is illustrated in solid black. The CPU utilization scale is provided on the right-hand side of the chart. These measurements were taken on a computer that was also running a variety of other non-Java tasks. The amount of CPU time taken by the other tasks is not reported directly in this chart. However, the impact of these other activities can be seen indirectly.

For example, the garbage collection effort spanning the time period from roughly 550 to 560 seconds has been preempted by some higher priority activity. During this time span, we see that the garbage collection effort lasts almost 10 seconds at a very low utilization of less than 10 percent following an initial burst of approximately 50 percent CPU utilization. During this same time span, the real-time Java application threads, which happen to run at a higher priority than the garbage collection thread in the measured configuration, are consuming less than 20 percent of the system CPU time.

From these observations, it is clear that the Java virtual machine's garbage collector has been preempted by other higher priority tasks running on the same computer. The ability to mix components written in different programming languages, as demonstrated with this example, is an important capability for mission-critical, real-time Java programming.

This particular application is running in a fairly predictable steady state as characterized by the following observations. First, the slope of the available memory chart is roughly constant whenever garbage collection is idle. This means the application's allocation rate is approximately constant. Second, with exception of the first peak, the heights of the available memory chart's peaks are roughly identical. This means the amount of live memory retained by the application is roughly constant. In other words, the application is allocating new objects at approximately the same rate it is discarding old objects. The reason the first peak is lower than the others is because other high-priority tasks in the system have preempted the garbage collector, delaying its completion. During the time that garbage collection is taking place, new objects con-

Figure 1: *Simulated Air Traffic Control Application With Paced Incremental Garbage Collection*



tinue to be allocated. Therefore, this peak is shorter than the others by roughly the amount of memory that was allocated during the extra wall-clock time required to complete this particular pass of the garbage collector.

Note that the percentage of CPU time consumed by real-time Java application threads is fairly predictable, ranging from about 10 percent to 50 percent, but is by no means constant. This is typical of real-world mission-critical systems. Most real systems exhibit variation in processing requirements as a result of fluctuations in the system workload. In systems that have real-time constraints, resources are budgeted conservatively to make sure there are enough resources to handle the occasional burst of demand for higher processing throughput.

Note that garbage collection is idle most of the time. As memory becomes scarce, garbage collection begins to run. In this example, garbage collection is configured to run at a lower priority than all of the real time application threads. When properly configured, the pacing agent will carefully avoid delaying the application threads by any more than the allowed scheduling jitter even when the garbage collection thread is configured to run at a priority higher than certain real-time Java threads.

Technology Adoption

The soft real-time garbage-collected Java technologies described in this article are commercially available in a cleanroom Java virtual machine product that conforms to the Java 2.0 Standard Edition (J2SE). The technologies have been commercially deployed in a number of mission-critical applications ranging from terabit-per-second fiber-optic switches to soft Programmable Logic Controller control of electric power generation and automation of semiconductor manufacturing. Together, these technology demonstrations represent over 100 developer-years of effort and have produced over a million lines of real-time Java code. Based on their experiences with these projects, developers have consistently found that they are much more productive and their software has fewer errors than when developing with C or C++. Some of their specific experiences are described in [10, 11].

The Nortel Optera HDX long-haul fiber-optic telecommunications switch provides an example of a recent application of real-time mission-critical Java. The hardware architecture for this product is fairly traditional. Redundant shelf controllers are combined in a large air-cooled chassis with a collection of line cards. In this product, both the *line cards* and the shelf controllers are based on PowerPC processors running a commercial real-time operating system.

The line cards have responsibility for the high-performance data transfer operations and implementation of communication protocol stacks. The shelf controllers have responsibility for managing and provisioning the resources contained on the line cards. The high-performance code that runs on the line cards is identified as control plane. The oversight software that runs on the shelf controllers is known as the management plane. Because shelf controllers need to communicate with the line cards, a small amount of management plane software runs on each of the line cards as well.

Previously, Nortel implemented the management plane software in C++; the most recent offering implements this functionality in Java for several reasons:

1. C++ is described as a *big language*, having many complex features that demand highly skilled developers and constant discipline to prevent creeping complexity making it difficult to maintain developed code economically.
2. Dynamic memory management problems in the earlier C++ implementation were particularly troublesome, leading to a variety of memory leaks, dangling pointers, and storage tramplers. Java's automatic garbage collection is to address these issues.
3. The management plane software that runs within the Optera HDX product must communicate with higher-level monitoring and supervisory components running on large Unix servers. In recent years, most of the Network Management System (NMS) and Element Management System (EMS) software running on those Unix servers has been replaced with Java technologies.

In considering whether to use real-time Java for the management plane components, Nortel engineers faced a variety of questions. For example, were the development tools mature enough to support efficient development? Could Java virtual machines run with sufficient reliability to assure the five nines reliability requirements common in the telecommunications industry? Would a real-time Java virtual machine be able to reliably support the 20-ms timing constraints that are imposed on certain management-plane reporting functions? Nortel engineers conducted an extensive yearlong evaluation of available Java technologies before making their final decision to adopt Java for this product.

The task of implementing the Optera HDX management plane software in Java took approximately two years with a development team comprised of more than 40 developers, resulting in a code base of more than a million lines of real-time Java code. Nortel has been selling the Optera HDX product since March 2002. The mission-critical Java components have since proven themselves in many months of successful service in hundreds of commercial deployments.

In evaluating their experience using Java, Nortel engineers have reported the following findings:

- Java software has been more reliable and Java developers have been more productive than their C++ counterparts.
- A large software module developed for the Optera HDX product was easily ported to another hardware platform for a related product.
- The object-oriented discipline utilized with Java made it possible to easily restructure the code to accommodate new requirements midway through development.
- Based on their successes with Java, Nortel intends to use more Java in next-generation products.
- Mistakes made by C programmers occasionally compromise the integrity of the Java virtual machine environment; thus there is motivation to develop lower level high-performance mission-critical Java to complement the soft real-time mission-critical Java that they have already deployed.

Standardization

The standards development being done by the Open Group's Real-time and Embedded Systems Forum [12] will establish a foundation that encourages competitive pricing and innovation among Java technology vendors while assuring portability and interoperability of real-time components written in the Java language. These standards, which are to be endorsed both by the Java Community Process and the International Organization for Standardization, will address a much broader set of requirements than the existing real-time specification for Java.

In particular, the standard for safety-critical Java will address concerns regarding certification under the Federal Aviation Administration's DO-178B guidelines. Beyond requirements for real time, the standard for mission-critical Java will address issues of portability, scalability, performance, memory footprint, abstraction, and encapsulation. Work within the Open Group is ongoing. The current plan is to deliver the safety-critical specification, refer-

ence implementation, and Technology Compatibility Kit by first quarter 2005. Working documents describing the Open Group's Real-time and Embedded Systems Forum's ongoing work standardization activities related to real-time Java are available at <www.opengroup.org/rtforum/rt_java> and <www.opengroup.org/rtforum/rt_safety>.

Table 1 summarizes key differences between different mission-critical Java technologies. The key points emphasized in this table are described in the following bulleted paragraphs:

- The standard J2SE Java libraries are keys to enabling high developer productivity, software portability, and ease of maintenance. Thus, it is important to provide all of these libraries to the soft real-time developer. Unfortunately, the standard J2SE libraries have a significant footprint requirement (at least four megabytes [Mbytes]) and depend heavily on automatic garbage collection, which is not available in the hard real-time environment. Thus, the hard real-time and safety-critical versions of Java cannot use the standard libraries. The hard real-time mission-critical Java standard will support the subset of the Connected Device Configuration libraries that is appropriate for a non-garbage-collected environment running on a limited-service, hard real-time executive. The safety-critical Java standard will support an even smaller library subset, pared down to facilitate safety certification efforts.
- The soft real-time mission-critical Java standard supports real-time garbage collection as described in this article. To improve throughput, determinism, and memory footprint requirements, the hard real-time and safety-critical Java standards do not offer automatic garbage collection.
- In traditional Java and soft real-time mission-critical Java, memory is reclaimed by garbage collection. There is no application programmer interface to allow developers to explicitly release objects, as this would decrease software reliability by introducing the possibility of dangling pointers. In the hard real-time mission-critical environment, we allow developers to explicitly reclaim the memory associated with certain objects. This is a dangerous service that must be used with great care. It is necessary, however, to support a breadth of real-world application requirements. In safety-critical systems, we prohibit manual deallocation of memory as use of this service would make it very difficult to certify safe operation of the software system.
- Traditional Java and soft real-time mission-critical Java allocate all objects in the heap. In the absence of automatic garbage collection, hard real-time and safety-critical Java developers can use special protocols to allocate certain objects on the run-time stack. The protocol includes compile-time enforcement of rules that assure that no pointers to these stack-allocated objects survive beyond the life-time of the objects themselves.
- Dynamic class loading allows new libraries and new application components to be loaded into a virtual machine environment on the fly. This is a very powerful capability, to be provided as broadly as possible. However, current safety certification practices are too restrictive to allow use of this capability in a safety-critical system.
- In the specification for traditional Java, thread priorities are mere suggestions. The virtual machine implementation may honor these suggestions, or it may ignore them. It may, for example, choose to treat all priorities with equal scheduling preference, or it may even choose to give greater scheduling preference to threads running at lower priorities. In all of the real-time Java specifications, priorities are distinct and priority ordering is strictly honored. The safety-critical Java specification implements strict first-in-first-out scheduling within priority levels, with no time slicing. This is the more common expectation for developers of safety-critical systems.
- Traditional Java does not offer any mechanism to avoid priority inversion, which might occur when a low-priority task locks a resource that is subsequently required by a high-priority task for it to make progress. The hard and soft real-time mission-critical specifications both support priority inheritance. Additionally, the hard real-time mission-critical Java standard and the safety-critical Java standard will support the priority ceiling protocol in which particular locks are assigned ceiling priorities, which represent maximum priority of any thread that is allowed to acquire this particular lock. Whenever a thread obtains a lock, its priority is automatical-

	Traditional Java	Mission-Critical Java		
		Soft Real Time	Hard Real Time	Safety Critical
Library Support	J2SE	J2SE	Subset of CDC	Very restrictive subset of CDC
Garbage Collection	Pauses in excess of 10 seconds	Real Time	No garbage collection	
Manual Memory Deallocation	Manual memory deallocation is disallowed		Allows manual deallocation	No manual deallocation
Stack Memory Allocation	No		Safe stack allocation	
Dynamic Class Loading	Yes			No
Thread Priorities	Unpredictable priority clustering and aging	Fixed priority, time-sliced preemptive, with distinct priorities		Fixed priority, distinct priorities, no time slicing
Priority Inversion Avoidance	None	Priority inheritance	Priority inheritance and priority ceiling	Priority ceiling
Asynchronous Transfer of Control	No	Yes		No
Approximate Performance	One-third to two-thirds speed of C	Within 10 percent of traditional Java speed	Within 10 percent of C speed	
Typical Memory Footprint	16+ Mbytes	16+ Mbytes	64 Kbytes - 1 Mbyte	64-128 Kbytes

Table 1: *Proposed Differentiation Between Java Technologies*

ly elevated to the ceiling priority level. If a thread with higher priority than the lock's ceiling priority attempts to acquire that lock, a run-time exception is generated. The priority ceiling mechanism is most efficient and is simpler to implement and to analyze for static systems in which all of the threads and their priorities are known before run time. The priority inheritance mechanism deals better with environments that experience dynamic adjustments to the thread population or to their respective priorities.

- Asynchronous transfer of control allows one thread to interrupt another in order to have that other thread execute a special asynchronous event handler and then either resume the work that had been preempted or abandon its current efforts. This capability, missing from traditional Java, is very useful in many real-time scenarios. We omit this capability from safety-critical systems because the asynchronous behavior is incompatible with accepted practices for safety certification.
- Because of the high-level services supported by Java, including automatic garbage collection, array subscript checking, dynamic class loading, and JIT compilation, traditional Java generally runs quite a bit slower than comparable algorithms implemented in, for example, the C language. Our experience with implementations of soft real-time Java is that they run a bit slower than traditional Java, because real-time garbage collection imposes a greater penalty on typical thread performance than non-real-time garbage collectors. The various compromises represented in the hard real-time and safety-critical Java specifications are designed to enable execution efficiency that is within 10 percent of typical C performance.
- Because of the size of the standard J2SE libraries and a just-in-time compiler, which is present in a typical J2SE deployment, the typical J2SE deployment requires at least 16 Mbytes of memory. Of this total, about half is available for application code and data structures. Depending on the needs of a particular application, the memory requirements may range much higher, up to hundreds of Mbytes for certain applications. Hard real-time mission critical Java is designed specifically to support very efficient deployment of low-level, hard real-time and performance-constrained software components. Though different applications exhibit different memory requirements, targeted applications typically run from about 64 kilobytes (Kbytes) up to a full Mbyte in memory requirements. Safety-critical deployments tend to be even smaller. This is because the costs of certification are so high per line of code that there is strong incentive to keep safety-critical systems as small as possible.

Conclusions

Increasingly, the military relies upon intelligence implemented as real-time software components to give their warfighters competitive advantage in modern conflicts. Developing and maintaining these large software systems represents tremendous cost and a high degree of risk. High-level programming languages like Java promise to reduce development and maintenance costs by two- to 10-fold, while increasing the reliability, flexibility, and functionality of embedded real-time systems.

Though early implementations of the Java virtual machine failed to address the needs of mission-critical real-time developers, newer technologies bring the full benefits of Java to this very important defense community.◆

References

1. Gosling, J., and H. McGilton. "The Java Language Environment: A White Paper." Mountain View, CA: Sun Microsystems, Inc., May 1996 <http://java.sun.com/docs/white/ langenv>.
2. Rovner, P. "On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically Checked Concurrent Language." Palo Alto, CA: Xerox Palo Alto Research Center, 1984 <www.parc.xerox.com/about/history/pub-historical.html>.
3. Nilsen, K. "Issues in the Design and Implementation of Real-Time Java." Real-Time Magazine Mar. 1998 <www.realtime-info.be/magazine/98q1/1998q1_p009.pdf>.
4. Nilsen, K. "Adding Real-Time Capabilities to the Java Programming Language." Communications of the ACM 41.6 (June 1998): 49-56 <http://doi.acm.org/10.1145/276609.276619>.
5. Klein, M., T. Ralya, B. Pollak, and R. Obenza. A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems. Kluwer Academic Publishers, Nov. 1993.
6. Bollella, G., J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. The Real-Time Specification for Java. Addison-Wesley, Jan. 2000.
7. J. Consortium's Real-Time Java Working Group, et al. "Real-Time Core Extensions." Cupertino, CA: J. Consortium. 2 Sept. 2000 <www.j-consortium.org/rtjwg/rtce.1.0.14. pdf>.
8. Nilsen, K., and A. Klein. Issues in the Design and Implementation of Efficient Interfaces Between Hard and Soft Real-Time Java Components. Proc. of the Workshop on Java Technologies for Real-Time and Embedded Systems. Catania, Sicily, Italy, 3-7 Nov. 2003.
9. Nilsen, K. Doing Firm Real-Time With J2SE APIs. Proc. of the Workshop on Java Technologies for Real-Time and Embedded Systems. Catania, Sicily, Italy, 3-7 Nov. 2003.
10. NewMonics, Inc. "Calix Success Story." Tucson, AZ: NewMonics, Inc., May 2003 <www.newmonics.com/perceval/calix_whitepaper. shtml>.
11. NewMonics, Inc. "Nortel Success Story." Tucson, AZ: NewMonics, Inc., Oct. 2003 <www.newmonics.com/perceval/nortel_whitepaper. shtml>.
12. The Open Group. Real-time and Embedded Systems Forum <www.opengroup.org/rtforum>.

About the Author

Kelvin Nilsen, Ph.D., is chief technology officer of Aonix, an international supplier of mission- and safety-critical software solutions. Nilsen oversees the design and implementation of the PERC real-time Java virtual machine along with other Aonix products, including Ameos MDA tools; ObjectAda compilers, development environment, and libraries; SmartKernel runtime executives; and commercial off-the-shelf safety certification support. Nilsen's pioneering research in real-time programming resulted in seven commercial patents. His seminal research on the topic of real-time Java led to the founding of NewMonics, a leader in advanced clean-room Java technologies. In 2003, Aonix acquired NewMonics. Nilsen has a Bachelor of Science in physics from Brigham Young University and a Master of Science and doctorate degree both in computer science from the University of Arizona.

Aonix
877 S. Alvernon WY STE 100
Tucson, AZ 85711
Phone: (520) 323-9011
Fax: (520) 323-9014
E-mail: kelvin@aonix.com

Separate Money Tubs Hurt Software Productivity

Dr. Ronald J. Leach
Howard University

Most software development organizations operate under four goals: more, better, cheaper, and faster. Reuse of existing software is often considered as a way to achieve these goals. Unfortunately, the project accounting practices of many organizations unwittingly discourage software project managers from improving costs and quality simultaneously. Here we show how a simple change in management practice and project accounting can encourage software development that meets these four goals. The simple change is consistent with the work of industry leaders such as Barry W. Boehm, David Weiss, James Coplien, Chi Tau Lai, and others on product line architectures.

In the November/December 1995 issue of the *Journal of Systems Management*, Paul Newcum [1] listed 13 problems that pervade the software industry:

1. Complexity.
2. Jargon.
3. Imprecise and inconsistent specifications.
4. Lack of up-front prototypes.
5. Lack of reusable software components.
6. Lack of realistic costs and schedules.
7. Difficulties using new paradigms.
8. Unrealistic deadlines.
9. Not removing defects and errors.
10. Quality not pursued.
11. Defects and errors regularly placed in software.
12. Poor business functions delivered initially.
13. Poor measurements of design and programming.

Any practicing software engineer, software project manager, or chief information officer can attest to the accuracy of Newcum's assessment. Unfortunately, the interaction between several of these problems is not as well understood as it should be. This article considers two of these – reuse and quality – and illustrates how some current management practices discourage possible quality improvement and cost savings. It then suggests simple changes that can help achieve software that meets all four objectives most organizations operate under: developing *more* software *better*, *cheaper*, and *faster*.

An Extremely Rosy Scenario

Suppose that an organization has two software projects, A and B. Suppose that half of the source code that is developed in project A also can be reused in project B. Suppose also, for the sake of simplicity, that the two projects are the same size and that the two projects interface smoothly, with no additional costs due to lack of imprecise or inconsistent specification standards (another on Newcum's list of pervasive software problems). Finally, assume that the portion of project A that is reused in project B does not require any changes.

If the manager of project A produces the software on time and within budget, and the software meets the predetermined quality standards (usually measured in the number of software defects per thousand lines of source code, or number of failures per thousand hours of operation of the software), then everyone is happy and he or she is likely to be rewarded.

> *"Most development organizations that have successful reuse programs recognize that there is overhead associated with reuse; this overhead is simply part of the developing organization's cost."*

What does this mean for the manager of project B? Suppose that he or she needs to reuse half of the source code from project A. Since only half of project B consists of new code, it is logical to assume that this project should have a smaller budget than project A. In most organizations, the determination of just how much smaller the budget should be depends upon the organization's experience with proper cost estimation for software projects with some amount of software reuse.

In this best of all possible worlds, the organization's cost estimation takes into account the amount of reuse and whether the requirements, design, or source code from project A are being reused in project B. (Earlier reuse is better than later, since the costs of all the remaining activities in the software life cycle of project B can be avoided from the point that the software is reused. There is no need to budget for requirements engineering, software design, coding, testing, or integration for any software that already exists.)

In this extremely rosy scenario, project A is produced on time, within budget, and with the expected level of quality; project B will also be produced on time, within budget, and with the level of quality expected by the organization. With perfect software reuse and extremely accurate cost estimation, project B has been created by a software development process that is both *cheaper* and *faster*. The high level of reuse has made the organization more efficient, giving us *more* software per month. It may even give us a *better* quality product for project B, because most of the software errors that normally occur during normal software development have already been removed (we hope) in project A.

Everything is wonderful. Or is it?

A Slightly Less Rosy Scenario

One problem that can occur even in this extremely rosy scenario is that project B may have a more stringent requirement for quality than did project A. For example, small errors that occur with improper capitalization of messages in a help system may not be worth fixing in a text editor. The quality of this system is probably sufficient, even with the error.

However, in a safety-critical application such as a user interface for a heart monitor, a confusing message can be the difference between life and death. An error in a seldom-used statistical routine can be ignored if it occurs in an inexpensive spreadsheet. The same error in software that controls the placement of coolant in a nuclear power plant can be disastrous. The problem in both cases is that software that is perfectly adequate for one application becomes dangerous when used in another. You can hear the legal team shuddering.

It appears that software reuse cannot always provide improvements in software quality, and in fact may degrade performance if integrated with higher quality components.

Clearly software reuse is dangerous, and can be expensive. Or is it?

A Solution

The difficulty here is that there is no incentive for project A to produce any higher quality of software than is needed for its requirements. The manager of project A views the budget as a tub of money, which can be dipped into to get project resources. The manager of project B has a similar view, with perhaps a different sized tub.

Many organizations use what some have called the *every-tub-on-its-bottom* approach to funding software projects. In this funding approach, the manager of a project is given a budget for completion of his or her project. The manager is rewarded for completion of the project under budget and within schedule, and held responsible to some extent if the project is either over budget or late (or both). The *tubs* of money are considered by project managers as resources to be used solely for their own projects.

If upper-level management follows the every-tub-on-its-bottom approach, then there is no incentive for the manager of project A to improve project quality to improve costs for another project. Even if the manager of project A decided to do so, there are no additional resources available to increase the quality of the product. In this case, the goals of more and better are directly in opposition to the goals of *cheaper* and *faster*.

The solution is for the developing organization to apply a small *reverse tax* to every software project that is likely to produce software that will be reused in another project. It is called a reverse tax because it is added to the budget of the project teams to allow them to provide the extra quality for contribution to a pool of reusable code. The *increased* funds provided in this reverse tax allow potentially reusable software to be given a quality check for its number of known errors, adherence to standards, documentation, and so on. The activities in this quality check are often referred to as certification in the software reuse literature. Certification of potentially reusable software can be paid by the reverse tax.

There are several questions that arise when considering the application of this reverse tax:

- **Who pays for this tax – the customer or the developer?** The development organization is responsible. Most development organizations that have successful reuse programs recognize that there is overhead associated with reuse; this overhead is simply part of the developing organization's cost. (Of course, to some degree, the customer always pays for the cost of development as part of the total software life-cycle cost, operational cost, and the true cost of having the software that is really needed by the customer's organization.)
- **How is it arranged?** Generally speaking, the development organization is responsible, although the customer may participate actively. Any organization considering a systematic approach to software reuse must do some *domain analysis* – the term used to describe, for example, determining how many additional projects are likely to need some portion of the current software [2, 3, 4]. Domain analysis is a generalization of systems analysis, in which the primary objective is to identify the operations and objects needed to specify information processing in a particular application domain. Domain analysis will precisely identify domains and software artifacts within these domains that are good candidates for reuse, and will estimate the economic benefits of reusing these software artifacts.
- **Who is doing this domain analysis?** The domain analysis is done by the development organization, in consultation with domain experts that may be outside consultants, members of the developer's internal staff, or even customer representatives.
- **Is there an overhead for systematic software reuse?** Of course there is overhead. Nothing is really free in the software industry. The overhead of systematic software reuse has been estimated at about 5 percent of overall cost, assuming that there is a metrics program, such as Capability Maturity Model® Level 2 or higher, in place [2].
- **Within the development organization, who pays for this overhead?** Other projects that are either concurrent with the selected projects, as well as future projects that will use the code pay for this tax. Particular projects that will create reusable components have some form of reverse tax added to their budgets for incorporating additional quality into these reusable components. The increased budget is intended to improve quality, not develop software from scratch.
- **Is there a net cost to the development organization?** There should be no net cost, provided that a reusable component that benefits from the *reverse tax* is actually reused.
- **Is there a net benefit to the development organization?** Yes, the net benefit is the difference between the cost of new development with a reused component versus the cost of new development. The cost savings increases greatly if the component is reused more than once.
- **How does the organization determine the appropriate amount of the reverse tax?** Additional testing and quality control measures (called certification) must be employed for each software artifact to be reused. The cost for this certification is gen-

"Particular projects that will create reusable components have some form of reverse tax added to their budgets for incorporating additional quality into these reusable components."

® The Capability Maturity Model is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

erally under 5 percent per reused artifact.

- **What is the potential effect on future projects?** They may be cheaper to develop, since not all code needs to be developed from scratch, and any reused code is certified as being of very high quality.
- **Are there any other potential problems with this accounting approach?** In some cases, there might be legal roadblocks. These roadblocks are unlikely, however, since many projects reusing software artifacts are written for the same customer.

Using the resources provided by this reverse tax allows a project to produce a higher quality system than it might do otherwise. If another project can reuse the higher quality source code that was produced by project A, then the initial extra cost due to the higher quality is recovered for the organization. Therefore, there is no additional cost from an organizational view.

As stated before, the situation changes for the better if there are several software development projects that can use the reusable code produced by project A. Improving project A's quality by reducing errors, improving documentation, and standardizing all software interfaces can simultaneously improve the quality and reduce the cost of all systems that reuse the source code from project A. This is clearly the way to get software projects that simultaneously achieve all four goals. Software development can be more, better, cheaper, and faster.

It is clear that the relatively simple institutional changes in accounting practice described in this article can make it possible for projects to improve both productivity and quality with a decrease in overall cost. The approach is essentially risk-free, because the reverse tax on any project is small, reducing any need for a major change in institutional practice and the inherent cultural risks associated with major institutional change. At the same time, this approach can help create a culture in which software reuse is enthusiastically adapted by all level of software engineers.

More Extensive Approaches

These ideas are, by no means, new. Barry Boehm [5] introduced the Win-Win approach, also known as *Theory W*, to software cost modeling and commercial off-the-shelf integration. His work subsumes the points made in this article. A more detailed discussion of incentives and disincentives to reuse can be found in [2].

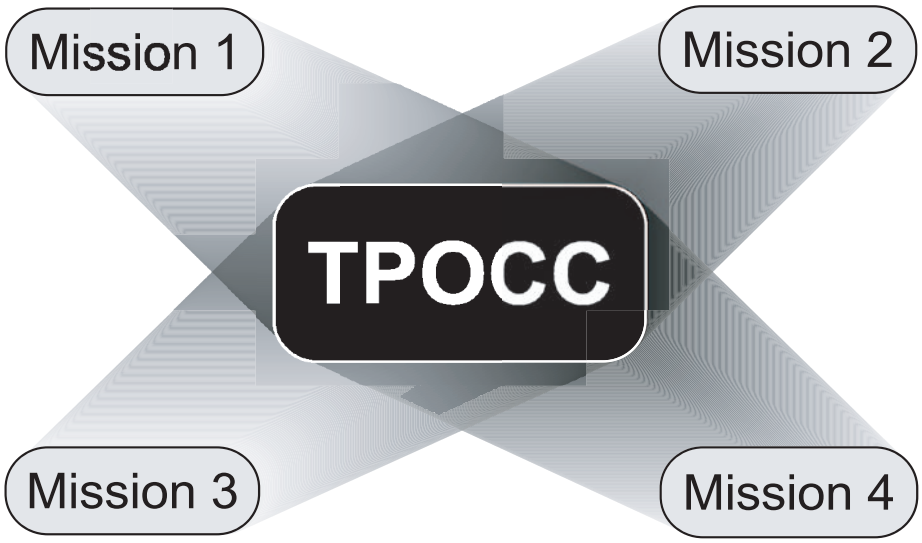


Figure 1: *Example of a Reusable Core of Spacecraft Control Software*

David Weiss and C. Lai [6] have written an important book on software product-line architectures. Much of their work is based on their experiences at Lucent Technologies and the resulting cost savings and quality improvement. An important follow-up paper [7] appeared in 1999. Note that the approach suggested here is much less formal than the complexity needed for the product-line architecture approach suggested by Weiss, Lai, and others.

Their work has been followed up by a series of publications on product-line architectures, including ones readily available from the Software Engineering Institute. Withey's report [8] is typical.

However, note that the ideas presented in this article have been used several places without the institutional reorganization needed to support a complete transition to a product-line architecture approach. For example, the author worked extensively on software for ground control of spacecraft at NASA Goddard Space Flight Center in Greenbelt, M.D., during a time of transition to a more reuse-based software development. The software team in what was then called the Control Center Systems Branch won a center-wide award for cost savings.

That branch was responsible for the ground systems that control the initial interface between a spacecraft and ground-based computer control centers. The control system software consists of large amounts of code organized into several subsystems to perform the following operations, among others:

- Determine the current position of the spacecraft.
- Control the operation of the spacecraft.
- Receive and relay telemetry information from the spacecraft.
- Detect significant events in spacecraft operation.
- Display the status of the system.

Space system software is extremely complex because it has severe requirements for fault tolerance, must interface with many other systems, and has some real-time requirements as well. An additional complexity is that the software must begin development far in advance of a projected launch of a spacecraft and therefore the level of technology of both hardware and support software (operating system, compilers, tools, commercial software, etc.) is not easy to determine during the beginning of development.

Reuse has been a concern for many years. However, the changing demands of spacecraft, the fluidity of graphics standards, the need for isolation from networks such as the Internet for security purposes, the long lead time for projects, and the need for severe restrictions on the weight of onboard computers all have made the development of a reuse program more difficult.

The initial step in any program of software reuse – domain analysis – was facilitated by a core group of talented domain experts, including both NASA employees and contractor personnel. The domain experts were already motivated by financial pressures and their desire to produce software in an efficient manner. They identified a reusable core of spacecraft control software (TPOCC in Figure 1)

and mission-specific software.

Accounting practices were modified on individual projects to incorporate the reverse tax to ensure that the TPOCC reusable core was of exceptionally high quality. There was little resistance, because the amount of work was overwhelming and any method to improve product quality and efficiency was accepted readily.

Examination of internal software discrepancy reports (*bug reports*) showed that the TPOCC reusable software core had several orders of magnitude errors fewer than other systems and subsystems, suggesting that the reverse tax funds saved by not duplicating software development had been properly allocated to improve quality of the most heavily reused components.

Other Benefits

Most of the problems that Newcum listed can be addressed by the simple change in project accounting proposed here. The apparent complexity of software projects is reduced by standard interfaces between component software *parts*. Client/server designs are consistent with high quality software with well-defined interfaces. Schedules can be made more realistic for projects that reuse high quality code, since there will be fewer problems integrating error-prone software with poorly specified interfaces. It is easier to support good business functions by reusing software components that are known to work.

It is less obvious, but equally true, that encouraging reuse by providing incentives to improve quality can improve design and encourage the use of up-front prototypes. Having a list of proven software components with standard interfaces can make the development of prototypes much faster.◆

Acknowledgement

This research was partially supported by the National Science Foundation under grant number 0324818.

References

1. Newcum, Paul. "13 Pains in My Software! With Healthy Medications for Each." Journal of Systems Management Nov./Dec. 1995: 28-31.
2. Leach, R.J. Software Reuse: Methods, Models, Costs. New York: McGraw-Hill, 1997.
3. Cohen, Sholom G., Jay L. Stanley Jr., A. Spencer Peterson, and Robert W. Krut Jr. "Application of Feature-Oriented Domain Analysis to the Army Movement Control Domain." Pittsburgh, PA: Software Engineering Institute, June 1992.
4. Prieto-Diaz, R. "Domain Analysis: An Introduction." Software Engineering Notes 15.2 (Apr. 1990): 47-54.
5. Boehm, B., P. Bose, E. Horowitz, and M.J. Lee. "Software Requirements Negotiation and Renegotiation Aids: A Theory-W Based Spiral Approach." International Conference on Software Engineering, Seattle, WA, Apr. 23-30, 1995.
6. Weiss, D.M., and C. Lai. Software Product Line Engineering. Addison Wesley Longman, New York, 1998.
7. Coplien, J., D. Hoffman, and D. Weiss. "Commonality and Variability in Software Engineering." IEEE Software Nov./Dec. 1999: 37-45.
8. Withey, J. "Investment Analysis of Software Assets for Product Lines." Pittsburgh, PA: Software Engineering Institute, Nov. 1996.

About the Author

Ronald J. Leach, Ph.D., is professor and chair of the Department of Systems and Computer Science at Howard University. Leach has had grants and contracts from many government agencies and companies and has given lectures on three continents. He does research in software engineering, with special interest in reuse, metrics, fault tolerance, performance modeling, process improvement, and the efficient development of complex software systems. He is the author of five books and more than 65 published technical articles. Leach has a Bachelor of Science, Master of Science, and doctorate degree in mathematics from Maryland University, and a Master of Science in computer science from Johns Hopkins University.

Department of Systems and Computer Science
School of Engineering
Howard University
Washington, D.C. 20059
Phone: (202) 806-6650
Fax: (202) 806-4531
E-mail: rjl@scs.howard.edu

Reuse and DO-178B Certified Software: Beginning With Reuse Basics

Hoyt Lougee
Foliage Software Systems

To successfully approach reuse and the associated certification considerations, a rigorous understanding of reuse is important. This article, from a certifiability perspective, defines reuse, discusses reuse drivers and typical reuse scenarios, and details the various types of reuse. In addition, a brief overview of a reuse analysis and implementation approach will be presented.

Reuse has been defined variously: Definitions range from "the systematic practice of developing software from a stock of building blocks, so that similarities in requirements and/or architecture between applications can be exploited to achieve substantial benefits in productivity, quality, and business performance" [1] to "the process of creating software systems from predefined software components" [2].

The first definition is seemingly more complete, but the second definition is less restrictive and more useful. Too often in literature, a purist attitude is taken toward reuse. For example, often the term reuse is applied to only those elements that can be used *without change* or to only those elements that have been designed and configured for reuse. For this article, therefore, reuse is defined simply as *using previously existing software artifacts*. Artifacts include all products of a certification development process and include planning data, requirements data, design data, source code, configuration management records, quality assurance records, and verification data.

Reuse Factors

Functional Alignment

Two aspects of functional alignment can affect the reuse strategy adopted. The first aspect, applicability, is a determination of how well the existing requirements/functionality align with the requirements of the target application. Do the artifacts serve the intended purpose? How much must the artifacts be modified to accommodate any new functionality? The second aspect also concerns the alignment of new functionality to existing functionality, although in the opposite respect. Does the existing configuration contain *more* functionality than is needed for the targeted application? What must be done to accommodate this additional functionality?

The issues surrounding extra functionality are prevalent with design-for-reuse component libraries. These libraries are designed to include all possible future functionality needs and, by their very nature, include additional functionality. These existing configurations typically contain *more* functionality than is needed for the targeted application. How, then, must this additional functionality be accommodated in a certifiable software system?

A variety of strategies are available to handle extra functionality. Seemingly, the most simple is to strip the unnecessary functionality from the configuration (from requirements through verification artifacts). Conceptually, this is the simplest approach, but this may not be the most cost-effective approach.

> *"Understanding the rigor with which previous development was performed is critical in determining the amount of effort that will be required to incorporate existing artifacts into a new configuration."*

Additional unused functionality may be retained in the code as long as the mechanism by which such code could be inadvertently executed is prevented, isolated, or eliminated [3] is verified. In other words, although the code is present, its non-availability within a specific application must be demonstrated. Typically, this means that the unused functional interface must be verified to ensure that the unused software is not used in a particular configuration.

These configuration mechanisms can entail a hardware switch such as jumper-pin settings, or can be performed purely in software. For example, if a software function is included in the object module but the entry point (the call to the particular routine) is not invoked, the software function can be shown as not accessed. Alternatively, if a routine includes a parameter switch to *turn off* parts of the routine's functionality, the switch mechanism can be verified and the software reviewed to ensure that the switch is always set appropriately.

Requirements Volatility

The ability to identify and isolate volatile requirements can maximize the ability to reuse. For example, if control logic were historically a primary source of change, an appropriate reuse strategy would dictate that the control logic is separated from non-volatile areas. This separation would enhance the ability to reuse non-volatile areas.

Both historical metrics as well as application-specific projections of change are important when considering requirements volatility. Applications may have inherent areas of instability that, by design, will always result in functional modifications; for example, application control laws that must be tuned for each targeted application. On the other hand, past areas of instability may have been resolved in the existing software baseline and application-specific changes indicating other *hot spots* are likely. In any event, a careful analysis of requirements volatility is vital in developing an appropriate reuse strategy.

Previous Development Rigor

Understanding the rigor with which previous development was performed is critical in determining the amount of effort that will be required to incorporate existing artifacts into a new configuration. When previous certification treatment is insufficient for the current application, whether the software to be reused is commercial off-the-shelf (COTS), software developed to other guidelines (for example, military guidelines), software certified to DO-178 or DO-178A, or software developed to DO-178B but to a lower software criticality level, effort must be expended to pro-

RTCA DO-178B: Software Considerations in Airborne Systems and Equipment Certification

Published by the Radio Technical Commission for Aeronautics, Inc. (RTCA) and adopted by the Federal Aviation Administration (FAA) Advisory Circular AC20-115B, DO-178B provides guidance in meeting airborne-product airworthiness requirements associated with software. Adherence to DO-178B adds an extra level of difficulty to the already challenging undertaking of embedded software development. The guidelines are not straightforward; interpretations vary, and acceptance is not always impartial.

DO-178B defines the objectives and activities that must be performed in developing and verifying airborne software systems. The specific objectives and the resulting rigor varies according to the criticality of the software, ranging from the most rigorous Level A for software whose failure can have catastrophic consequences to Level E for software whose failure has no effect on the aircraft's continued safe flight and landing.

Adherence to DO-178B, therefore, will produce evidence by which the applicant can instill confidence in the FAA that the software embedded in airborne equipment is safe for its intended use. The software development and verification processes necessary to generate this evidence can be costly and time consuming. As a consequence, avionics manufacturers, struggling with their cost and schedule constraints, often turn to reuse.

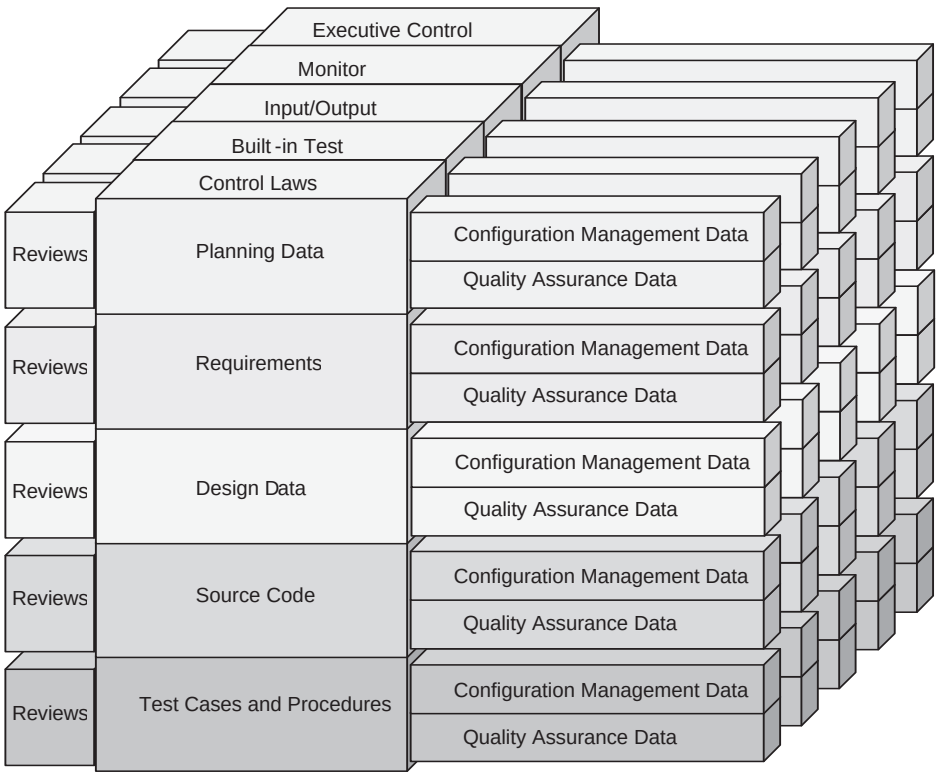
vide assurance that the software is suitable for the target certification effort.

If the previous software development was not certified with DO-178B, the existing development artifacts must be analyzed and mapped to the objectives of DO-178B. As a guideline, DO-178B does not dictate specifics with respect to data items or specific development processes. Instead, DO-178B details objectives that must be satisfied. Often, especially with military applications, a great deal of rigor was applied to the development process and a wealth of reusable artifacts is available.

On the other hand, if the previous development was certified with DO-178B, the previous development criticality level will determine the types of artifacts created, how the artifacts were configured and the type of change control provided, and the extent to which verification was performed.

Verification independence is also driven by the software criticality level. Higher criticality levels require greater levels of independence; therefore, the impacts of resolving independence issues must be considered. For example, criticality levels A and B require independence in assuring that the software high-level requirements comply with the system requirements and that the high-level requirements are accurate and consistent. These reviews must be re-addressed if the requirements are to be reused.

Figure 1: *Example Structured-Design-Based Configuration*



Maturity of Existing Artifacts

As a rule of thumb, reuse of *buggy* code is not a good idea – especially if the functionality is to be modified. Debugging modifications of *buggy* code compounds the complexity of the development process. As software issues arise during development, the source of the issues is not clear. Was the problem related to recent changes, was the problem related to reused elements, or was the problem related to a combination of the two? Moreover, the pedigree of buggy code may not be clear: Some bugs may necessitate major architectural changes – changes that, unfortunately, were not factored into the initial reuse analysis.

Defect history can be analyzed in several ways to *get a feel for the bugginess* of the previous software. The number and character of the defects found in the previous development effort can identify problem areas and provide insight into the amount and types of problems that can be expected. Analysis of the overall defect trending is also important. If the software was released and the defect-identification rate was still increasing, the software is sure to have undetected defects. On the other hand, if the defect-find rate was asymptotically approaching zero defects, the probability of a large number of undetected defects is lower.

Targeted Platform Changes

Often the ability to reuse software and reap the initial considerable investment in certifiable-software development is hampered by changing hardware platforms. Platforms change for many reasons ranging from strategic technology migration to obsolescence issues. Regardless of the motivation for changing the platform, the effects on software reuse are critical to the overall project impact.

Too often, the decision to update the hardware platform is performed without considering software reuse – disastrous effects on project schedules and budgets typically result. Since software development increasingly requires the lion's share

of project budgets, software reuse should be a central consideration when developing a hardware migration strategy.

The target platforms must be analyzed in terms of concurrent multiple-platform support and the anticipated platform life span. The overlying product/business strategy must be examined to determine the need to support multiple-concurrent platforms. Questions to ask include the following:

- Is the software intended for use on varying concurrent platforms?
- What is the anticipated life span of targeted platforms?
- Is there a hardware migration plan (and if not, why not)?
- What are the characteristics of anticipated future platforms?
- Will future platforms be based on the same family of processors?
- Will the same basic hardware design/interface be retained?

Reuse Strategies

Full Vertical Reuse Versus Partial Vertical Reuse

Reuse can entail the entire life-cycle artifact set or subset. Full vertical reuse includes all life-cycle artifacts related to specific functionality. With certifiable aviation-software configurations, vertical reuse would entail all software life-cycle data: planning data, requirements data, design data, verification data, as well as configuration management data and quality assurance data. Partial vertical reuse would include a subset of this data: Perhaps only the requirements and design data would be appropriate for reuse. Clearly, full vertical reuse is preferred, but significant cost and schedule savings can still be accomplished by analyzing existing software systems for partial vertical reuse opportunities. Figure 1 illustrates a simple configuration based on a structured design.

As shown in Figure 2, full vertical reuse includes all life-cycle artifacts of the development, whereas in this example, partial vertical reuse only includes the requirements and design. Note that the associated quality assurance data and configuration management data, as well as the associated review data, are included with each vertical layer.

Even with full vertical reuse, however, there is still work to be performed to incorporate the reuse within a new application. Suppose that a feature whose functionality is unchanged is to be reused. Furthermore, suppose that all associated life-cycle data is expected to be accurate and appropriate with respect to the targeted application. A finite amount of work must still be performed and documented to ensure that the artifacts are indeed appropriate for use in the targeted application.

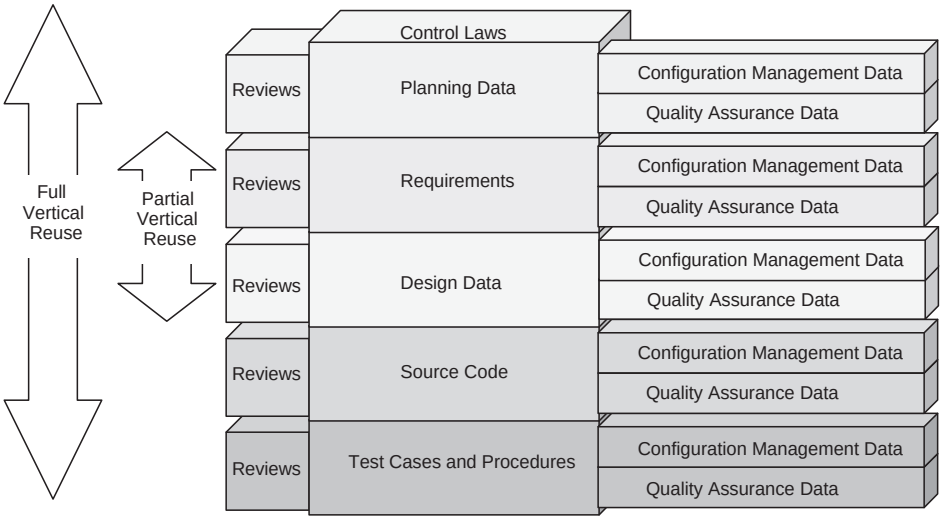


Figure 2: *Partial Versus Full Vertical Reuse*

With all this extra activity, what is gained? The reviews and analyses performed are typically neither as extensive nor time consuming as the initial *from scratch* reviews. These reviews and analyses are specific and focused on the integration of the reused artifacts into the target application. With respect to the design/architecture and code reviews, the focus is on the external interface to the reusable functionality. If the code to be reused includes 150 complex modules, only two of which interface externally, only the two modules would be the subjects of in-depth review.

Note that all previous and new review and analyses evidence are appropriate for the new certification effort.

Full Horizontal Versus Partial Horizontal Reuse

Reuse can include all artifacts within a specific life-cycle step or a subset. Full horizontal reuse includes all artifacts within a specific life-cycle step as illustrated in Figure 3. For example, reusing all source code would be an example of full horizontal reuse: All functionality is appropriate for the new application. Partial horizontal reuse would entail the extraction of a subset of functionality.

Typically, most design-for-reuse artifact libraries currently used today are horizontal code-component libraries. Often, an overall design is created based on the requirements at hand and an understanding of what is available in the code-component library. In fact, code libraries provided by language vendors adhere to this model: The user is to create an application-specific design based on the requirements and the language capabilities to support varying designs. With aviation software reuse, however, the common designs are critical in creating families of applications, especially with respect to alignment to hardware architectures.

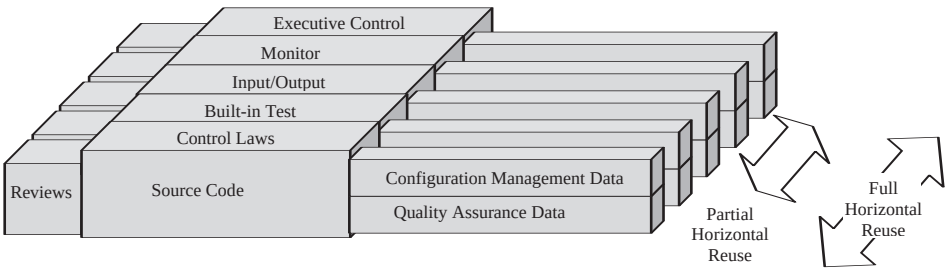
Note that partial horizontal reuse might not be undertaken with the goal of enhancing the systems features; partial horizontal reuse may be used to remove extraneous functions to create simplified applications.

Finally, as with vertical reuse, the verification of the reuse component interface to the target hardware and software is key. Careful analysis and planning for these interfaces must be performed.

Designed for Reuse Versus Not Designed For Reuse

Many organizations approach reuse from the purist *design-for-reuse* point of view.

Figure 3: *Partial Versus Full Horizontal Reuse*



These organizations create reusable artifacts with the express purpose of populating reuse repositories for use in future applications. These reusable artifacts typically require more time to create because they must be functionally robust to accommodate all expectations for future usage.

Once created, these repositories inevitably suffer from functionality creep: The *ultimate* functionality provided often does not account for some future usage scenario. As a result, the reusable elements are either duplicated and modified resulting in two similar elements to sustain or the element is extended with care for backwards compatibility. Of course, design for reuse can be very valuable and often provides the greatest cost savings. Clearly, well-conceptualized artifacts are easier to sustain and extend than constrained artifacts (designed as intended for initial use). But just as often, the costs associated with the creation of the repository are underestimated, as is the volatility of the functionality desired.

On the other hand, organizations often employ *scavenge* reuse, that is, harvesting existing artifacts that were not specifically designed for reuse. Depending upon the initial quality of the artifact, as well as the amount of horizontal and vertical reuse appropriate, scavenging existing software artifacts is often the best solution. On the other hand, if the initial software suffers from quality issues or the *fit* within the target application is not clean, starting from scratch may be the appropriate strategy.

Not Modified for Reuse Versus Modified for Reuse

Software artifacts that need not be modified for reuse typically offer the fewest certification hurdles. The cost-benefit is highest with scenarios in which the previous data can be used *as is* and only the applicability and interfaces verified. Minimal changes to artifacts, therefore, result in minimal additional certification effort. Often, only a regression analysis and a minimal regression suite are required to accommodate changed artifacts.

Changes to artifacts should be well considered to minimize the impact. Requirements and architectural changes in the software in which the reusable artifact is to be incorporated should often be tailored around reusable artifacts to minimize the overall project cost and schedule. Careful analysis of the requirements, design, and architecture of both the configuration to be reused, as well as the configuration into which reusable artifacts are to be incorporated, can provide critical input into the cost/benefit analysis and reuse strategy selection.

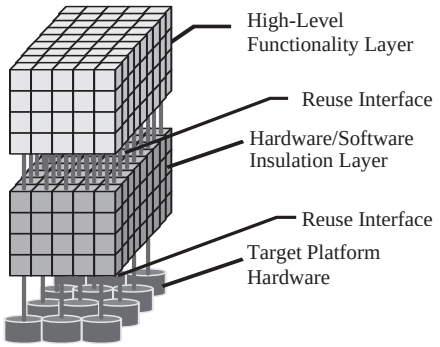
Partitioned Versus Non-Partitioned Reuse

Partitioned [4] software provides natural divisions for horizontal reuse. Designs that can partition software into volatile and non-volatile elements and minimize the amount of interfaces to be verified can result in significant cost and schedule savings. Since higher levels of criticality drive higher costs and longer schedules, minimizing the amount of software with critical functionality is desirable.

Partitioning a software system to separate higher- and lower-criticality levels can minimize the more costly critical severity verification activities. If the critical software partition is further designed with reuse in mind, the benefits can be twofold. For example, if engine-control software is partitioned into critical built-in test and engine-control functionality versus noncritical built-in test and monitoring communications, reuse could be performed on each partition independently.

If the noncritical built-in test and monitoring is most volatile, the more expensive engine-control and critical built-in test partition need not be completely re-addressed each time the more volatile areas change. This reused software can also build service history, lowering certification risk, and increasing confidence in the overall application.

Figure 4: *Layered Architectural Approach*



Reuse Scenarios

Common Functionality – Different Target Platform

Avionics manufacturers often mitigate the effects of changing platforms with a layered architectural approach (see Figure 4). This architecture provides for a hardware-interface layer to insulate high-level application software from the effects of changing platforms: Software accommodation of hardware changes is limited to this interface layer. For different applications with different functionality using the same hardware, this insulation layer remains constant and can be reused. For applications with different hardware but with the same functionality, this interface layer cannot be reused, but the application software that *sits* on top of the insulating layer can be reused.

In so far as an application is certified for the functionality provided and that functionality depends on both the insulation layer and the application layer, the interface between the insulation layer and the application layer must be verified when either the high-level functionality layer or the insulation layer changes.

Common Functionality – Different Tools

Different target platforms, especially those not in the same family, often require changes in the toolset used in development and verification of the software system. Manufacturers often resist changing toolsets because of the additional impacts on tool qualification and new-tool learning curves. These hidden costs are often neglected in the planning of product changes.

Changes in design methodology typically have greater impact on qualification than do simple changes in compiler versions or changes in language. Isolation of change impact and the extent/scope of the regression analysis are typically more extensive with design methodology changes. Changing both the design methodology and the language compounds reuse issues.

With different languages, versions of the same language (and associated toolset changes), or even when different sets of compiler options are to be used that would result in different object code, previous DO-178B verification activities are typically invalidated and must be re-performed. When a different processor is used, the development toolset and the resulting object code will necessarily change, even for the same source code. In addition, hardware/software integration verification must be repeated. Hardware/software integration tests and hardware/software compatibility reviews must be updated as appropriate and performed again.

Common Functionality – Different Development Standards

When previous certification treatment is insufficient for the current application, whether the software to be reused is COTS, software developed to other guidelines (for example, military guidelines), software certified to DO-178 or DO-178A, or software developed to DO-178B but to a lower software criticality

level, special certification considerations can be invoked.

In all cases, the objectives of DO-178B must be satisfied completely. The system safety assessment for the new application will provide guidance in the level to which this certification effort must proceed. Typically, reuse of significant portions of existing artifacts (code and supporting documentation) can be leveraged. Reverse engineering may be used to regenerate software life-cycle data that is inadequate or missing. As with all types of reuse, the Plan for Software Aspects of Certification should detail the strategy to ensure early buy-in by the certification authorities.

Common Functionality – Refactoring

For the purposes of this discussion, refactoring concerns the update of the software design and implementation without necessarily changing the functionality, tools, or target environment. Because of the significant costs of certification, refactoring is not typically performed; the rationale that "if it's not broken, don't fix it" provides the most cost/benefit. However, sometimes refactoring is necessary to provide a more robust, sustainable, and/or extensible application – and may be the appropriate, long-term, strategic approach.

Different Functionality – Common Platform/Tools

If functional alignment is not exact, change must be accommodated as discussed previously. This situation is probably the most common reuse scenario and includes both modifications of a single application for defect resolution and functional enhancements, and feature tailoring for different versions of the same application.

First and foremost in accommodating functional changes is the isolation of the change area within the software architecture. Localized changes facilitate regression verification, especially if specifically designed to do so. In addition, certification authorities, when reviewing modifications to certifiable configurations, must understand the change impact. Clearly defined localized changes are easier to analyze, easier to document, and easier to communicate to the certification authority.

Localizing changes is facilitated with highly cohesive and loosely coupled reusable artifacts. Loosely coupled elements have low levels of interdependency with their environments. Highly cohesive elements have a high level of uniformity in the element's functional goal. A cohesive element serves one functional purpose.

Aviation and Obsolescence

Electronic component manufacturers base their offerings on the ability of their product lines to generate income. Income is generated by selling quantities of components at specific prices. These manufacturers maximize their profits by providing high quantities at low cost rather than low quantities at higher cost. Unfortunately, the aviation market is not a significant consumer of electronic components. Instead, cell phones and personal computers drive the market offerings. As such, the fast, competitive pace of technology evolution has increasingly affected aviation manufacturers. Careful planning of component provisioning and migration are key to a successful product strategy.

Obsolescence issues make reuse key. In recent years, the obsolescence of electronic components has driven avionics manufacturers to redesign for new processors, memory and communications chips, and other electronic components.

Design for reuse often becomes design for obsolescence as manufacturers strive to reduce the cost of fielding their products. Manufacturing companies are often faced with a dilemma: Can a sufficient supply of components be purchased as a *last time* buy for all future projected use, can secondary suppliers that create the component in lower quantities at exorbitant prices supply the components, or should a redesign be performed with the associated recertification issues? Redesign often provides the ability to incorporate new functionality and defray future obsolescence issues, but the cost is often prohibitive. Reuse of all or parts of the software configuration can be key to finding the least-risk, go-forward manufacturing approach.

Different Functionality – Different Platform/Tools (Portability)

Changing functionality, as well as the target platform/toolsets, often occurs as new families of products are developed. A modular approach can be adopted to address the changes.

In contrast to the layered architectural approach, dividing architecture by specific functionality is also advantageous. As requirements change over time or as different members of the same application family require different functionality, specific functional divisions can be advantageous. The granularity of the architectural divisions must be carefully considered to isolate areas that can change independently (non-cohesive elements). In the extreme, hardware and software become reusable components – a *plug-and-play* strategy can be adopted. The best-known example of this reuse scenario is, of course, the personal computer with the wide array of associated peripheral devices. As with horizontal layering, verification of the interface is key to the incorporation of reusable elements.

Same Application – Different Aircraft Installation

DO-178B provides for "airborne systems or equipment, containing software that has been previously certified at a certain software level and under a specific certification basis" [3], being used in new aircraft installations.

If the system safety assessment performed for the new system does not indicate a new software level, the software configuration may be used *as is*. All software artifacts may be provided without change or further work for the new application certification. The development of a new Plan for Software Aspects of Certification and Accomplishment Summary may be the only tasks to be performed.

If any changes are to be performed to accommodate the new aircraft installation, the software must be treated as indicated above for previously developed software.

Benefits of Reuse

Technical and commercial trade journals widely tout reuse's promised benefits. Lower development costs, reduced programmatic risks, and shortened schedules flow from successful reuse and result in enhanced corporate competitive advantage. The value of reuse increases as the time, cost, and expertise invested in product development are continually leveraged across an ever-wider range of products.

A less publicized benefit of reuse is safety. For example, the concept of building a service history illustrates one of the safety benefits of reuse. Reused software, when properly analyzed and integrated, undergoes greater scrutiny and *time on the wing* over time. Another reuse safety benefit derives from properly partitioned software, which minimizes the amount of interaction between non-critical and critical software, thereby reducing the number of possible error sources for critical functionality.

A final benefit of reuse derives from

defect reporting over multiple applications. A well-coordinated reuse strategy will track defects common to reused components. When a defect is found on an application in a reuse component, other applications that use the same reuse component can be examined for defect impact and updated as appropriate.

Industry Case Study: Primus Epic

Honeywell's Primus Epic illustrates many of the reuse concepts discussed above. To address industry demands for system scalability, system reliability and maintainability, and reduced acquisition and application costs, Primus Epic, Honeywell's next generation integrated avionics systems, incorporates a highly flexible and cost-efficient framework.

The Primus Epic Product Line Architecture (PLA), packages integrated modular units and line-replaceable units into a single aircraft-wide Virtual Backplane Network. This architecture allows data generated by each system component to be available to all other system components.

Variation in the PLA is supported by the Module Avionics Unit (MAU): a cabinet containing field replaceable modules. These building blocks provide input/output, processing, and database storage functions.

The building blocks housed in the MAU communicate using the Avionics Standard Communications Bus using a Network Interface Controller module [5].

Partitioning was used with great effect to separate hardware components, separate hardware from software components, and separate software components within the Primus Epic system. Honeywell, for example, received a Federal Aviation Administration Technical Standard Order (TSO) approval for the modular avionics cabinet as an item of hardware. The various avionics functions such as the flight management system contained in the software are obtained with a separate TSO. The certification impacts of subsequent development or changes to a particular established configuration are considerably reduced.

Honeywell's Digital Engine Operating System (DEOS), which forms the Primus Epic software platform, provides standard services and interfaces for hosted applications. This operating system supports RTCA DO-178B partitioning, which minimizes the certification costs of the hosted software application by allowing different certification levels for applications with different criticality considerations. Different software functions hosted on DEOS can be certified with varying amounts of rigor, depending upon their particular effects on safety. Since software development and certification costs have increased astronomically in relation to the hardware costs, this capability further supports cost, schedule, and risk savings [6].

Honeywell's success with their Primus Epic system illustrates well the variability possible with a solid, common PLA. Primus Epic serves as the foundation for business jet, regional aircraft, and helicopter cockpits, including Dassault's Enhanced Avionics System (EASy) cockpit to be used in all new Falcon Jet models. The PLA accommodates extensive variation, including changes in aircraft configurations, changes in aircraft integrating components, radically different look and feel for both the displays (from two to six displays), and a variety of user input devices (from traditional controllers to new cursor control devices and voice-command mechanisms). Moreover, the specific functionality supported can range from movable navigation maps and real-time video to engine instrument and crew advisory systems and primary flight and navigation systems.

Primus Epic was designed as an integration platform; consider the EASy flight deck, which is based upon Primus Epic. Dassault was the primary system architect and worked in cooperation with Honeywell to create EASy. Honeywell opened up their previously proprietary communications bus specification to enable the creation or modification of a variety of custom and off-the-shelf components. Dassault was able to select among compliant avionics vendors to populate the cockpit, create new and effective configurations, and maintain their competitive advantage [7].

Conclusion

Cost and schedule can be saved, and safety can be enhanced with reuse for DO-178B certifiable software. A thorough understanding of the key reuse factors, a clear purpose and goals, a solid analysis, and careful planning are necessary to maximize the benefits of reuse. Manufacturers must analyze the many types of reuse and select among them. Reuse is a complex endeavor and the benefits are only available to those who approach it with care.◆

References

1. Ezran, M., M. Morisio, and C. Tully. Practical Software Reuse. 1st ed. London: Springer-Verlag, 15 Feb. 2002.
2. Dr. Carma McClure. "Model-Driven Software Reuse Practing Reuse Information Engineering Style," 1995 Extended Intelligence, Inc.
3. Radio Technical Commission for Aeronautics, Inc. RTCA DO-178B, Software Considerations in Airborne Systems and Equipment Certification. Section 4.4.3 Structural Coverage Analysis Resolution, (d) Deactivated Code. Washington, D.C.: RTCA, 1 Dec. 1992 <www.rtca.org>.
4. Radio Technical Commission for Aeronautics, Inc. RTCA DO-178B, Software Considerations in Airborne Systems and Equipment Certification. Section 2.3.1 Partitioning. Washington, D.C.: RTCA, 1 Dec. 1992 <www.rtca.org>.
5. Honeywell. "Primus Epic: Topology." Feb. 1998 <www.myflite.com/myflite/products/ias/epic/14T0p040gy.jsp>.
6. Hughes, David. "This Is Not Deja Vu." Aviation Week & Space Technology 4 June 2003.
7. Taverna, Michael A. "Making Flight Easy." Aviation Week & Space Technology 2 June 2003.

About the Author

Hoyt Lougee is the engineering manager, Aviation Division, at Foliage Software Systems. Foliage delivers software architecture, custom software development, and technology strategy consulting. Lougee's responsibilities include program management and software process improvement. Previously with AlliedSignal/Honeywell, Lougee has more than 13 years of experience with both military (DoD-STD-2167a) and commercial (RTCA DO-178B) aviation software development and certification efforts. Lougee has authored a number of white papers and presented at the 2002 Digital Avionics Systems Conference.

Foliage Software Systems
168 Middlesex TPKE
Burlington, MA 01803
Phone: (781) 993-5500
Fax: (781) 993-5501
E-mail: hlougee@foliage.com

CROSSTALK
The Journal of Defense Software Engineering

ARTICLE INDEX VOLUME 17

TOPIC	ARTICLE TITLE	AUTHOR(S)	ISSUE	PAGE
Project Management	Catastrophe Disentaglement: Getting Software Projects Back On Track	E.M. Bennatan	10	10
	Software Project Management Practices: Failure Versus Success	Capers Jones	10	5
Quality	Common Errors in Large Software Development Projects	David A. Gaitros	3	21
	Right Sizing Quality Assurance	Walt Lipke	7	25
Requirements	Better Communication Through Better Requirements	Michael J. Hillelsohn	4	24
	Requirements Engineering So Things Don't Get Ugly	Deb Jacobs	10	19
	Understanding Software Requirements Using Pathfinder Networks	Udai K. Kudikyala, Dr. Rayford B. Vaughn Jr.	5	21
Reuse	Applying Decision Analysis to Component Reuse Assessment	Michael S. Russell	4	20
	An Economic Analysis of Software Reuse	Dr. Randall W. Jensen	12	4
	Estimating and Managing Project Scope for Maintenance and Reuse Projects	William Roetzheim	12	9
	Reuse and DO-178B Certified Software: Beginning With Reuse Basics	Hoyt Lougee	12	23
	Separate Money Tubs Hurt Software Productivity	Dr. Ronald J. Leach	12	19
	Using Java for Reusable Embedded Real-Time Component Libraries	Dr. Kelvin Nilsen	12	13
Risk Management	A Project Risk Metric	Robert W. Ferguson	4	12
	Risk Factor: Confronting the Risks That Impact Software Project Success	Theron R. Leishman, Dr. David A. Cook	5	31
Security	Advanced Software Technologies for Protecting America	Gregory S. Shelton, Randy Case, Louis P. DiPalma, Dan Nash	5	10
	Competitiveness Versus Security	Don O'Neill	6	24
	A Survey of Anti-Tamper Technologies	Dr. Mikhail J. Atallah, Eric D. Bryant, Dr. Martin R. Stytz	11	12
Software Consultants and Mentors	Lessons Learned From Software Engineering Consulting	Dr. David A. Cook, Theron R. Leishman	2	4
	Ten Key Techniques for Effective Consulting in a Challenging Environment	Sarah A. Sheard, Suzanne Zampella, Albert J. Truesdale	2	11
	Verification and Validation People Can Be More Than Technical Advisors	George Jackelen	2	26
Software Development	Object-Oriented Layers in ELIST	Mary Ann Widing, Kathy Lee Simunich, Dariusz Blachowicz, Mary Braun, Dr. Charles Van Groningen	1	23
Software Edge	Service-Oriented Architechture and the C4ISR Framework	Dr. Yun-Tung Lau	9	11
	Software Wars	Susan Weaver	9	4
	Tomahawk Cruise Missile Control: Providing the Right Tools to the Warfighter	Marcus Urioste	9	8
Software Estimation	Extreme Software Cost Estimating	Dr. Randall W. Jensen	1	27
Software Inspections	When Is It Cost Effective to Use Formal Software Inspections?	Bob McCann	3	30
Systems Engineering	Enterprise Composition	John Wunder	8	27
	Managing Requirements for a System of Systems	Ivy Hooks	8	27
	A Recommended Practice for Software Reliability	Dr. Norman F. Schneidewind	8	13
Team Software Process	Using the Team Software Process in an Outsourcing Environment	Miguel A. Serrano, Carlos Montes de Oca	3	9
Testing	Efficient and Effective Testing of Multiple COTS-Intensive Systems	Dr. Richard Bechtold	5	26
	Introducing TPAM: Test Process Assessment Model	Dr. Yuri Chernak	6	30
Top 5 Articles	2004 U.S. Government's Top 5 Programs	David R. Castellano	10	4
	The Advanced Field Artillery Tactical Data System Proves Successful in Battle	Pamela Palmer	7	6
	CrossTalk Honors the 2003 Top 5 Quality Software Project Finalists	Pamela Palmer	7	18
	The DMLSS Program Brings Electronic Commerce to the Military Treatment Facilities	Pamela Palmer	7	8
	The H1E System Configuration Set Lays the Foundation for Decades to Come	Pamela Palmer	7	10
	The One-SAF Objective System Fits Individual Simulation Needs	Chelene Fortier-Lozancich	7	12
	Patriot Excalibur Software Enables Full-Scale Deployment of Battle-Ready Units	Chelene Fortier-Lozancich	7	14
	Winning Projects Exemplify Success for Developers and Acquirers	Elizabeth Starrett	7	4
Training	Overcoming Training Dilemmas Brings Greater Training Value	Gregory T. Daich	2	7

CONTINUED ON NEXT PAGE

Hey Buddy – Need a Fix?

The basis for this column is an event that happened to my family – and it started out as a far cry from anything computer related. My daughter had a sports injury last summer. She is a distance runner on her school's track team. As luck would have it, there was a sports-medicine specialist at our hospital. We first visited our primary provider, who referred her to the sports-medicine specialist. We visited the specialist, who gave amazingly useful advice to my daughter on how to cure and prevent this type of injury in the future. As we were leaving the doctor, he told us to be sure to make a follow-up appointment with him in two or three weeks.

After a few weeks had passed, I called the local appointment line to make my daughter's follow-up appointment. I was told that she couldn't make an appointment with the specialist until she once again saw her regular provider, and he made yet another consultation referral request. I realized the wasted effort of arguing with the appointment clerk (who, it was apparent, was reading off of a prepared script). However, I pointed out to the clerk that my daughter's prescriptions were to expire shortly, and an appointment and referral would take another week. It was just as futile asking her help with the prescription refill dilemma. Finally, I gave up and asked, "Who can I talk to that will be able to handle my problem?" She gave me the name and number of Fred (name changed to protect the innocent).

I quickly called Fred, who agreed that the appointment process would not let us make a direct follow-up with a specialist, only with our primary care provider. However, Fred said that he had ways around the problem. Two clicks on the computer, and BANG! It now appeared that the specialist was my daughter's primary care provider. The computer, now happily digesting this piece of (incorrect) data, scheduled an appointment for us with the specialist for later on in the day. After all the appointments with the specialist were over, all we had to do was call Fred one more time, and he would adjust the database so that the specialist was no longer listed as the primary care provider.

Breathing a sigh of relief, I realized that I had just talked to a fixer. A fixer is a person who is able to fix a broken or unwieldy process and make it meet user needs. Now, as I am sure you will recognize, a process evolves over time. The appointment process at our hospital originally allowed the appointment clerks to make appointments with any doctor. However, too many people abused the system by requesting a specialist without a prior consultation with a general practice doctor. ("Hello, appointment clerk? I have a headache, and I just know I have terminal brain cancer, so please schedule me an appointment with a neurosurgeon.") So the new system totally prohibits appointments to specialists. The problem is that there are some times when special actions have to be taken in violation of the current process. Enter the role of a fixer.

There is nothing wrong with having a fixer, as long as the fixer is able to make the process better over time. Fred freely admitted that the current process was too restrictive, and told my wife and me to drop by when we were at the hospital and fill out a patient advocate action request form (formerly known as a complaint form). Fred said he helps collect and review these forms, and they use them to suggest changes to the current process.

Does your process have flaws? Of course it does! Can you find a way around them yourself? If so, then you need to work to make the process more workable. If you can't find a way around the process yourself, you might have to resort to a fixer. Just make sure that the fixer is working to constantly improve the current process. If the fixer just fixes and moves on the process will become more and more broken over time.

— David A. Cook, Ph.D.
Senior Research Scientist
The AEgis Technologies Group, Inc.
dcook@aegistg.com

P.S. By the way, CROSSTALK is always looking for BACKTALK authors. If interested, e-mail and we'll *fix* you right up.

MONTHLY COLUMNS:

ISSUE		COLUMN TITLE	AUTHOR
Issue 1: January Information From Senior Leadership	Publisher: BackTalk:	Changes Come to CrossTalk and to Warfare Operations Software Insecta Zodiac	H. Bruce Allgood Gary Petersen
Issue 2: February Consultants	Publisher: BackTalk:	Finding the Right Consultant Brings Mutual Success Laws of Software Motion	Elizabeth Starrett Dr. David A. Cook, Theron Leishman
Issue 3: March Software Process Improvement	Publisher: BackTalk:	Buying and Building Systems and Software Better Who Moved My Job?	Tracy Stauder Gary Petersen
Issue 4: April Acquisition	Publisher: BackTalk:	Contract Oversight Requires Data Software Process Improvement: A Good Idea for Other People	Elizabeth Starrett Dr. David A. Cook
Issue 5: May Technology: Protecting America	Publisher: BackTalk:	Technology By Any Other Name The Technology of War	Brent Baxter Gary Petersen
Issue 6: June Assessments and Certifications	Publisher: BackTalk:	Know Where Your Organization Stands Today and Determine How to Improve for Tomorrow I'm Sorry Dave – I Can't Certify That	Tracy Stauder Dr. David A. Cook
Issue 7: July Top 5	Publisher: BackTalk:	2003 U.S. Government's Top 5 Quality Software Projects Movie Physics and the Software Industry	David R. Castellano Gary Petersen
Issue 8: August Orchestrating a Systems Approach	Publisher: BackTalk:	Defining Systems Stone Software Development	Elizabeth Starrett Robert K. Smith
Issue 9: September The Software Edge	Publisher: BackTalk:	Greater Combat Effectiveness Systems Engineering and Shoe Polish	Tracy Stauder Dr. David A. Cook
Issue 10: October Project Management	Publisher: BackTalk:	CrossTalk Welcomes New Sponsors A little Learning is a dang'rous Thing; Drink deep, or taste not the Piereian Spring	Tracy Stauder Barry Schrimsher
Issue 11: November Software Toolbox	Publisher: BackTalk:	What's in Your Toolbox? Broken Windows	Elizabeth Starrett Gary Petersen
Issue 12: December Reuse	Publisher: BackTalk:	Reuse: A Maturing Practice Hey Buddy – Need A Fix?	Tracy Stauder Dr. David A. Cook

CrossTalk / MASE
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

PRSRT STD
U.S. POSTAGE PAID
Albuquerque, NM
Permit 737